Driver Libraries Help

MPLAB Harmony Integrated Software Framework

© 2013-2015 Microchip Technology Inc. All rights reserved.

Driver Library Overview

This topic provides an overview of the MPLAB Harmony Driver Library.

Introduction

A device driver provides a simple well-defined interface to a hardware peripheral that can be used without operating system support or that can be easily ported to a variety of operating systems.

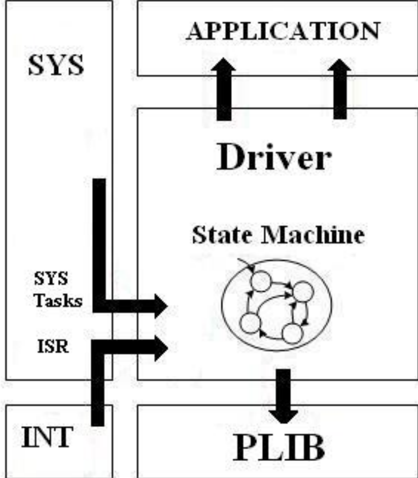
Description

The basic driver operations allow an application to interact with a device, reading and writing data, as if it was a simple file. More specific operations are present on most drivers and the kind of specific operations available depends on the peripheral whose functionality is being exposed by the driver. A driver has the following fundamental responsibilities:

- Providing a highly-abstracted interface to a peripheral
- Controlling access to a peripheral
- Managing the state of a peripheral

The following diagram illustrates how a driver interacts with the other pieces of the system:

- The application calls the well defined driver interface to use the services provided by the driver
- The driver calls various system services to perform the tasks that are possibly shared across other drivers
- The driver also calls the peripheral library of the peripheral to which it is abstracting the interface
- Driver State machine can be invoked by the system task service (polling system) or the driver state machine can be invoked from an ISR.



The driver provided the interfaces which can be broken down into the following two categories, System Operation and Client Operation *(<mod> is the abbreviation identifying the module).*

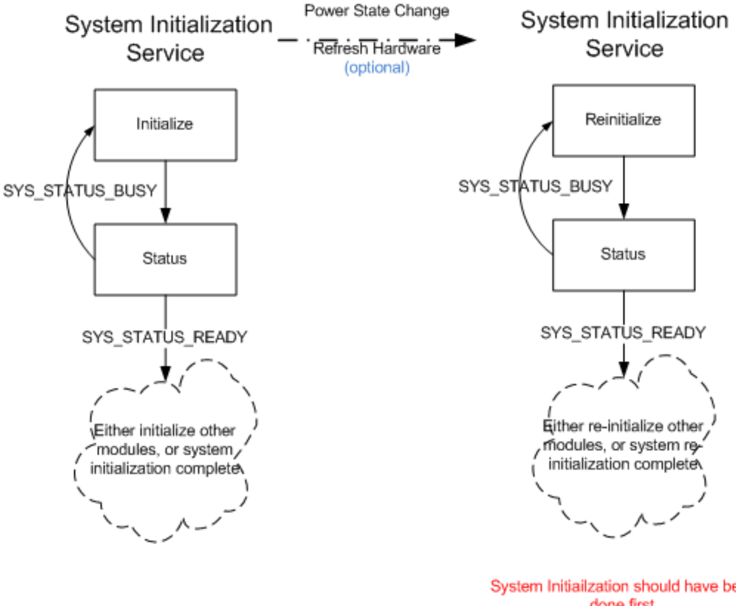
System Operation

The interfaces for the system operation should be called by the system. Each driver can support all of the system interfaces, or it can choose to not support the optional system interfaces. The system interfaces are:

- **DRV_<mod>_Initialize** - This routine initializes hardware for the index instance of the <module> module, using the hardware initialization given data. It also initializes any internal data structures. The DRV_<mod>_Status operation will return SYS_STATUS_READY when this operation has completed. Every driver module should define its own initialization data structure type named "DRV_<mod>_INIT". This structure must be an extension of the SYS_MODULE_INIT structure (i.e., its first member must be the SYS_MODULE_INIT structure or equivalent). Any parameter that can change the power state of the module must be included in the data structure. Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.
- **DRV_<mod>_Reinitialize** - This routine reinitializes and refreshes the hardware using the hardware initialization given data. The DRV_<mod>_Status operation will return SYS_STATUS_READY when this operation has completed. It does not clear or reinitialize internal data structures (although it may change the value of a few appropriate data items necessary to manage the new hardware state) and it does not disconnect or interrupt any ongoing client operations. This operation can be used to change the power state of the peripheral the module manages.

- **DRV_<mod>_Deinitialize** - This routine deinitializes the index instance of the <module> module, disabling its operation (and any hardware for driver modules).The DRV_<mod>_Status operation will return SYS_STATUS_READY when this operation has completed.
- **DRV_<mod>_Status** - The Status operation can be used to determine when any of the other three module operations has completed. If the Status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the Status operation returns "SYS_STATUS_READY", any previous operations have completed.
  - The value of SYS_STATUS_READY is 1. Values between 1 and 10 are reserved for system defined "ready" states. A module may define module-specific "ready" or "run" states greater than or equal to 10 (SYS_STATUS_READY_EXTENDED).
  - The value of SYS_STATUS_ERROR is -1. Values between -1 and -10 are reserved for system-defined errors. A module may define module-specific error values of less than or equal to -10 (SYS_STATUS_ERROR_EXTENDED).
  - If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations. (Remember to check the value returned by the Status routine after calling any of the module operations to find out when they have completed.)
- **DRV_<mod>_Tasks** - Used to maintain the driver's state machine and implement its ISR. It is normally only called by the system's tasks routine or by an Interrupt Service Routine (ISR).

The usage model of the system interfaces is demonstrated in the following diagram:



The system is also responsible for either calling the Tasks routine through an ISR or poll the Tasks routine in a polling environment.

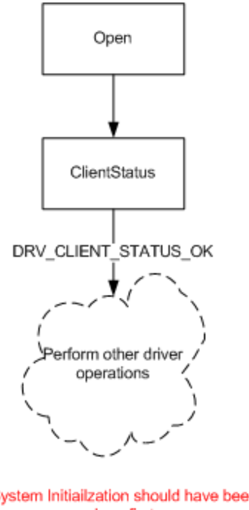
Client Operation

The interfaces provided by the driver for client operation, can be called by the application. Each driver can provide a set of client interfaces, which supports the operation model of the peripheral that the driver is supporting. The client interfaces can include:

- **DRV_<mod>_Open** - This routine opens a driver for use by any client module and provides an "open-instance" handle that must be provided to any of the other driver operations to identify the caller and the instance of the driver/hardware module.
- **DRV_<mod>_Close** - This routine closes an opened-instance of a driver, invalidating the given handle. This routine is optional, if the driver is designed to never be closed.
- **DRV_<mod>_ClientStatus** - This routine provides the client, the status of the driver.
- **DRV_<mod>_<operation> -** This is the basic format of a driver interface routine. Driver interface routines (other than the system module routines and the basic device driver routines) should follow this format.

The general usage model of the client interfaces is demonstrated in the following diagram, each driver will document

usage model specific to the driver in the driver help file.



Driver Configuration

A driver provides a set of configurations which can be divided into following categories:

- **Build Configuration Selection Configurations** - The driver supports selecting the driver instance at build time (static configuration) or at run time (dynamic configuration). A driver also supports selecting the number of clients that can possibly connect to an hardware instance. All drivers will provide DRV_<mod>_DRIVER_OBJECTS_NUMBER, when this configuration macro is defined driver is built in dynamic driver configuration, else it is built in static driver configuration. The number assigned to DRV_<mod>_DRIVER_OBJECTS_NUMBER controls the maximum number of driver objects that can be used. If a driver supports multiple clients it will provide DRV_<mod>_CLIENT_OBJECTS_NUMBER, when this configuration macro is defined driver is built to support multiple clients, else the driver is built to support single client. The number assigned to DRV_<mod>_DRIVER_OBJECTS_NUMBER controls the maximum number of client objects that can be used.
- **Initialization Overrides** - Initialization overrides are provided to enable configuration of the hardware statically and with low run time overhead. These override the parameters if also passed dynamically to Initialize or Reinitialize routine.
- **Other operational configurations** - These can control the functionality of the driver or other setting of the driver that are required for support core or optional functionality.

A driver allows the application to use the dynamic interface for all the previously described configurations when the application chooses the appropriate configurations at build time. This is the preferred method of usage for the drivers.

As an exception, if the application wants to use multiple configurations in the same build for a driver type, or it needs to use multiple instances of the same driver type, it needs to directly use the static single - client configuration interfaces, or it needs to use the static multiple - client configuration interfaces.

While using the driver the static single client driver configuration, the application and system can choose to ignore the SYS_MODULE_INDEX, SYS_MODULE_OBJ and DRV_HANDLE passed and returned to the driver and system interfaces.

While using the driver the static multiple client driver configuration, the application and system can choose to ignore the SYS_MODULE_INDEX and SYS_MODULE_OBJ passed and returned to the driver and system interfaces.

Library Interface

Constants

	Name	Description
	DRV_CONFIG_NOT_SUPPORTED	Not supported configuration.
	DRV_HANDLE_INVALID	Invalid device handle.
	DRV_IO_ISBLOCKING	Returns if the I/O intent provided is blocking
	DRV_IO_ISEXCLUSIVE	Returns if the I/O intent provided is non-blocking.
	DRV_IO_ISNONBLOCKING	Returns if the I/O intent provided is non-blocking.
	_DRV_COMMON_H	This is macro _DRV_COMMON_H.
	_PLIB_UNSUPPORTED	Abstracts the use of the unsupported attribute defined by the compiler.

Data Types

	Name	Description
	DRV_CLIENT_STATUS	Identifies the current status/state of a client's connection to a driver.
	DRV_HANDLE	Handle to an opened device driver.
	DRV_IO_BUFFER_TYPES	Identifies to which buffer a device operation will apply.
	DRV_IO_INTENT	Identifies the intended usage of the device when it is opened.

Description

This section describes the common Application Programming Interface (API) functions of the Driver Libraries.

Refer to each section for a detailed description.

Data Types

DRV_CLIENT_STATUS Enumeration

Identifies the current status/state of a client's connection to a driver.

File

driver_common.h

C

```
typedef enum {
  DRV_CLIENT_STATUS_ERROR_EXTENDED = -10,
  DRV_CLIENT_STATUS_ERROR = -1,
  DRV_CLIENT_STATUS_CLOSED = 0,
  DRV_CLIENT_STATUS_BUSY = 1,
  DRV_CLIENT_STATUS_READY = 2,
  DRV_CLIENT_STATUS_READY_EXTENDED = 10
} DRV_CLIENT_STATUS;
```

Members

Members	Description
DRV_CLIENT_STATUS_ERROR_EXTENDED = -10	Indicates that a driver-specific error has occurred.
DRV_CLIENT_STATUS_ERROR = -1	An unspecified error has occurred.
DRV_CLIENT_STATUS_CLOSED = 0	The driver is closed, no operations for this client are ongoing, and/or the given handle is invalid.
DRV_CLIENT_STATUS_BUSY = 1	The driver is currently busy and cannot start additional operations.
DRV_CLIENT_STATUS_READY = 2	The module is running and ready for additional operations
DRV_CLIENT_STATUS_READY_EXTENDED = 10	Indicates that the module is in a driver-specific ready/run state.

Description

Driver Client Status

This enumeration identifies the current status/state of a client's link to a driver.

Remarks

The enumeration used as the return type for the client-level status routines defined by each device driver or system module (for example, DRV_USART_ClientStatus) must be based on the values in this enumeration.

DRV_HANDLE Type

Handle to an opened device driver.

File

driver_common.h

C

```
typedef uintptr_t DRV_HANDLE;
```

Description

Device Handle

This handle identifies the open instance of a device driver. It must be passed to all other driver routines (except the initialization, deinitialization, or power routines) to identify the caller.

Remarks

Every application or module that wants to use a driver must first call the driver's open routine. This is the only routine that is absolutely required for every driver.

If a driver is unable to allow an additional module to use it, it must then return the special value DRV_HANDLE_INVALID. Callers should check the handle returned for this value to ensure this value was not returned before attempting to call any other driver routines using the handle.

DRV_IO_BUFFER_TYPES Enumeration

Identifies to which buffer a device operation will apply.

File

driver_common.h

C

```
typedef enum {
  DRV_IO_BUFFER_TYPE_NONE = 0x00,
  DRV_IO_BUFFER_TYPE_READ = 0x01,
  DRV_IO_BUFFER_TYPE_WRITE = 0x02,
  DRV_IO_BUFFER_TYPE_RW = DRV_IO_BUFFER_TYPE_READ|DRV_IO_BUFFER_TYPE_WRITE
} DRV_IO_BUFFER_TYPES;
```

Members

Members	Description
DRV_IO_BUFFER_TYPE_NONE = 0x00	Operation does not apply to any buffer
DRV_IO_BUFFER_TYPE_READ = 0x01	Operation applies to read buffer
DRV_IO_BUFFER_TYPE_WRITE = 0x02	Operation applies to write buffer
DRV_IO_BUFFER_TYPE_RW = DRV_IO_BUFFER_TYPE_READ|DRV_IO_BUFFER_TYPE_WRITE	Operation applies to both read and write buffers

Description

Device Driver IO Buffer Identifier

This enumeration identifies to which buffer (read, write, both, or neither) a device operation will apply. This is used for "flush" (or similar) operations.

DRV_IO_INTENT Enumeration

Identifies the intended usage of the device when it is opened.

File

driver_common.h

C

```
typedef enum {
  DRV_IO_INTENT_READ,
  DRV_IO_INTENT_WRITE,
  DRV_IO_INTENT_READWRITE,
  DRV_IO_INTENT_BLOCKING,
  DRV_IO_INTENT_NONBLOCKING,
  DRV_IO_INTENT_EXCLUSIVE,
  DRV_IO_INTENT_SHARED
} DRV_IO_INTENT;
```

Members

Members	Description
DRV_IO_INTENT_READ	Read
DRV_IO_INTENT_WRITE	Write
DRV_IO_INTENT_READWRITE	Read and Write
DRV_IO_INTENT_BLOCKING	The driver will block and will return when the operation is complete
DRV_IO_INTENT_NONBLOCKING	The driver will return immediately
DRV_IO_INTENT_EXCLUSIVE	The driver will support only one client at a time
DRV_IO_INTENT_SHARED	The driver will support multiple clients at a time

Description

Device Driver I/O Intent

This enumeration identifies the intended usage of the device when the caller opens the device. It identifies the desired behavior of the device driver for the following:

- Blocking or non-blocking I/O behavior (do I/O calls such as read and write block until the operation is finished or do they return immediately and require the caller to call another routine to check the status of the operation)
- Support reading and/or writing of data from/to the device
- Identify the buffering behavior (sometimes called "double buffering" of the driver. Indicates if the driver should maintain its own read/write buffers and copy data to/from these buffers to/from the caller's buffers.
- Identify the DMA behavior of the peripheral

Remarks

The buffer allocation method is not identified by this enumeration. Buffers can be allocated statically at build time, dynamically at run-time, or even allocated by the caller and passed to the driver for its own usage if a driver-specific routine is provided for such. This choice is left to the design of the individual driver and is considered part of its interface.

These values can be considered "flags". One selection from each of the groups below can be ORed together to create the complete value passed to the driver's open routine.

Constants

DRV_CONFIG_NOT_SUPPORTED Macro

Not supported configuration.

File

driver_common.h

C

```
#define DRV_CONFIG_NOT_SUPPORTED (((unsigned short) -1))
```

Description

Not supported configuration

If the configuration option is not supported on an instance of the peripheral, use this macro to equate to that configuration. This option should be listed as a possible value in the description of that configuration option.

DRV_HANDLE_INVALID Macro

Invalid device handle.

File

driver_common.h

C

```
#define DRV_HANDLE_INVALID (((DRV_HANDLE) -1))
```

Description

Invalid Device Handle

If a driver is unable to allow an additional module to use it, it must then return the special value DRV_HANDLE_INVALID. Callers should check the handle returned for this value to ensure this value was not returned before attempting to call any other driver routines using the handle.

Remarks

None.

DRV_IO_ISBLOCKING Macro

Returns if the I/O intent provided is blocking

File

driver_common.h

C

```
#define DRV_IO_ISBLOCKING(intent) (intent & DRV_IO_INTENT_BLOCKING)
```

Description

Device Driver Blocking Status Macro

This macro returns if the I/O intent provided is blocking.

Remarks

None.

DRV_IO_ISEXCLUSIVE Macro

Returns if the I/O intent provided is non-blocking.

File

driver_common.h

C

```
#define DRV_IO_ISEXCLUSIVE(intent) (intent & DRV_IO_INTENT_EXCLUSIVE)
```

Description

Device Driver Exclusive Status Macro

This macro returns if the I/O intent provided is non-blocking.

Remarks

None.

DRV_IO_ISNONBLOCKING Macro

Returns if the I/O intent provided is non-blocking.

File

driver_common.h

C

```
#define DRV_IO_ISNONBLOCKING(intent) (intent & DRV_IO_INTENT_NONBLOCKING )
```

Description

Device Driver Non Blocking Status Macro

This macro returns if the I/ intent provided is non-blocking.

Remarks

None.

_DRV_COMMON_H Macro

File

driver_common.h

C

```
#define _DRV_COMMON_H
```

Description

This is macro _DRV_COMMON_H.

_PLIB_UNSUPPORTED Macro

Abstracts the use of the unsupported attribute defined by the compiler.

File

driver_common.h

C

```
#define _PLIB_UNSUPPORTED
```

Description

Unsupported Attribute Abstraction

This macro nulls the definition of the _PLIB_UNSUPPORTED macro, to support compilation of the drivers for all different variants.

Remarks

None.

Example

void _PLIB_UNSUPPORTED PLIB_USART_Enable(USART_MODULE_ID index);

This function will not generate a compiler error if the interface is not defined for the selected device.

Files

Files

Name	Description
driver.h	This file aggregates all of the driver library interface headers.
driver_common.h	This file defines the common macros and definitions used by the driver definition and implementation headers.

Description

This section lists the header files used by the Driver libraries.

driver.h

This file aggregates all of the driver library interface headers.

Description

Driver Library Interface Header Definitions

Driver Library Interface Header This file aggregates all of the driver library interface headers so client code only needs to include this one single header to obtain prototypes and definitions for the interfaces to all driver libraries. A device driver provides a simple well-defined interface to a hardware peripheral that can be used without operating system support or that can be easily ported to a variety of operating systems. A driver has the fundamental responsibilities:

- Providing a highly abstracted interface to a peripheral
- Controlling access to a peripheral
- Managing the state of a peripheral

Remarks

The directory in which this file resides should be added to the compiler's search path for header files.

File Name

drv.h

Company

Microchip Technology Inc.

driver_common.h

This file defines the common macros and definitions used by the driver definition and implementation headers.

Enumerations

	Name	Description
	DRV_CLIENT_STATUS	Identifies the current status/state of a client's connection to a driver.
	DRV_IO_BUFFER_TYPES	Identifies to which buffer a device operation will apply.
	DRV_IO_INTENT	Identifies the intended usage of the device when it is opened.

Macros

	Name	Description
	_DRV_COMMON_H	This is macro _DRV_COMMON_H.
	_PLIB_UNSUPPORTED	Abstracts the use of the unsupported attribute defined by the compiler.

	DRV_CONFIG_NOT_SUPPORTED	Not supported configuration.
	DRV_HANDLE_INVALID	Invalid device handle.
	DRV_IO_ISBLOCKING	Returns if the I/O intent provided is blocking
	DRV_IO_ISEXCLUSIVE	Returns if the I/O intent provided is non-blocking.
	DRV_IO_ISNONBLOCKING	Returns if the I/O intent provided is non-blocking.

Types

	Name	Description
	DRV_HANDLE	Handle to an opened device driver.

Description

Driver Common Header Definitions

This file defines the common macros and definitions used by the driver definition and the implementation header.

Remarks

The directory in which this file resides should be added to the compiler's search path for header files.

File Name

drv_common.h

Company

Microchip Technology Inc.

ADC Driver Library

This topic describes the Analog-to-Digital Converter (ADC) Driver Library.

Introduction

This Analog-to-Digital Converter (ADC) driver provides an interface to manage the ADC module on the Microchip family of microcontrollers.

Description

An ADC is a vital part of any system that interfaces to real-world signals. While there are many techniques for analog-to-digital conversion, the Microchip family of microcontrollers uses Successive Approximation as one of its primary techniques.

Note: Only Static implementation is supported for the ADC Driver Library.

Using the Library

This topic describes the basic architecture of the ADC Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_adc.h`

The interface to the ADC Driver Library is defined in the `drv_adc.h` header file.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the ADC Driver Library.

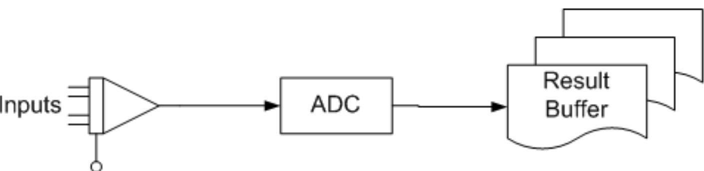
Section	Description
Configuring the Library	Provides macros for configuring the system. It is required that the system configures the driver to build correctly by choosing appropriate configuration options as listed in this section.
System Interaction Functions	Provides system module interfaces, Device initialization, deinitialization, reinitialization and status functions.
Client Core Configuration Functions	Provides interfaces for core functionality of the driver.
Other Functions	Provides additional ADC Driver functions.

Abstraction Model

This library provides a low-level abstraction of the Analog-to-Digital (ADC) Driver Library on Microchip's microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The ADC driver is modeled using the abstraction model, as shown in the following diagram.



- DRV_ADC_InputsRegister allows selection of inputs
- DRV_ADC_SamplesRead and DRV_ADC_SamplesReadLatest allows reading of the sample from the result buffer
- DRV_ADC_Start and DRV_ADC_Stop will control the operation of the ADC

How the Library Works

The library provides interfaces to support:

- System Interaction
- Client Core Functionality

System Initialization

This section describes the system initialization and reinitialization settings for the ADC Driver Library.

Description

System Initialization and Reinitialization

The system initialization and the reinitialization settings, affect only the instance of the peripheral that is being initialized or reinitialized. During system initialization configure each instance of the module with the following configuration settings that are supported by the specific ADC device hardware (refer to DRV_ADC_INIT).

1. Device requested power state: One of the system module power states.
2. Acquisition time: If the hardware supports this feature, configure this variable from DRV_ADC_ACQUISITION_TIME.
3. Voltage reference: If the hardware supports this feature, select the one from the available options(from DRV_ADC_VOLTAGE_REFERENCE).
4. Conversion clock: If the hardware supports this feature, select the conversion clock prescaler value (from DRV_ADC_CONVERSION_CLOCK_PRESCALER).
5. Conversion clock source: If the hardware supports this feature, select the conversion clock prescaler value (from DRV_ADC_CONVERSION_CLOCK_SOURCE).
6. Clock frequency: Peripheral clock frequency configured for the device.
7. Conversion trigger source: If the hardware supports this feature, select the conversion trigger source(from DRV_ADC_CONVERSION_TRIGGER_SOURCE).
8. Samples per interrupt: Configure the number of samples before to be generating interrupt(from DRV_ADC_SAMPLES_PER_INTERRUPT).
9. Output data format: Select the output data format.
10. Interrupt source: Select the interrupt source.

Example: Auto Sampling Mode

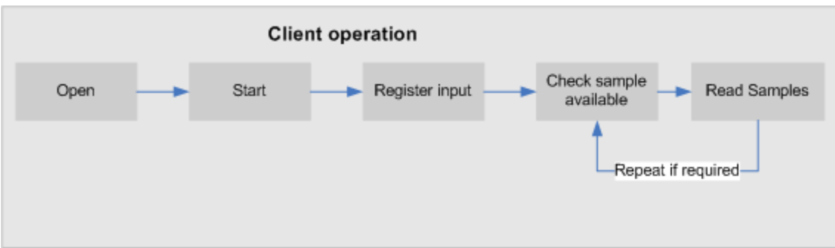
```
#define MY_ADC_INSTANCE DRV_ADC_INDEX_0

    SYS_MODULE_OBJ     myAdcObj;
    DRV_ADC_INIT       adcInitData;
    SYS_STATUS         adcStatus;

    // Populate the adcInitData structure
    adcInitData.plibModuleId = ADC_ID_1;
    adcInitData.acquisitionTime = PLIB_ADC_ACQUISITION_TIME_20_TAD;
    adcInitData.voltageReference = PLIB_ADC_VREF_POS_TO_VDD_VREF_NEG_TO_VSS;
    adcInitData.clockFrequency = 4000000; //4MHz
    adcInitData.conversionClock = PLIB_ADC_CONV_CLOCK_20_TCY;
    adcInitData.conversionClockSource = PLIB_ADC_CLOCK_SRC_SYSTEM_CLOCK;
    adcInitData.conversionTriggerSource = PLIB_ADC_CONVERSION_TRIGGER_INTERNAL_COUNT;
    adcInitData.dataOutputFormat = PLIB_ADC_OUTPUT_FORMAT_INTEGER_16BIT;
    adcInitData.initFlags = DRV_ADC_AUTO_SAMPLING;
    adcInitData.interruptSource = PLIB_INT_SOURCE_ADC_1;
    adcInitData.samplesPerInterrupt = PLIB_ADC_SAMPLE_PER_INTERRUPT_AT_EACH_SAMPLE;

    myAdcObj = DRV_ADC_Initialize(MY_ADC_INSTANCE, (SYS_MODULE_INIT*)&adcInitData);

    adcStatus = DRV_ADC_Status(myAdcObj);
    if (SYS_STATUS_BUSY == adcStatus)
    {

        // do something else and check back later
    }
    else if (SYS_STATUS_ERROR >= adcStatus)
    {
        // Handle error
    }
```

Client Core Functionality

Core functionality provides a basic interface for the driver operation.

Description

General Client Operation



Core functionality provides a basic interface for the driver operation.

Applications using the timer core functionality, need to perform the following:

1. The system should have completed necessary initialization and DRV_ADC_Tasks should either be running in a polled environment, or in an interrupt environment.
2. Open the driver using DRV_ADC_Open (the timer driver supports exclusive access only).
3. Registers the inputs to be used by the clients using DRV_ADC_InputsRegister.
4. Start the driver using DRV_ADC_Start.
5. Poll for the elapse status using DRV_ADC_SamplesAvailable, and then read the samples using either DRV_ADC_SamplesReadLatest or DRV_ADC_SamplesRead.
6. The client will be able to stop the started ADC instance using DRV_ADC_Stop at any point and will be able to close it using DRV_ADC_Close when it is no longer required.

Example:

```
DRV_HANDLE adcHandle;        //Handle returned by DRV_ADC_Open function.
uint16_t dataBuffer;         //Buffer to which the data will be written.

/*
Open a new client. The handle returned by the this function should be used
a passing parameter for all the specific client related operations.
*/
adcHandle = DRV_ADC_Open(MY_ADC_INSTANCE, DRV_IO_INTENT_NONBLOCKING);

DRV_ADC_InputsRegister ( adcHandle ,PLIB_ADC_INPUT_AN1|PLIB_ADC_INPUT_AN2 );

/*
ADC is not yet enabled. Enable it once the input set is registered. Starts automatically
since auto sample mode is enabled.
*/
DRV_ADC_Start(adcHandle);

if ( DRV_ADC_SamplesAvailable(adcHandle) )
{

    DRV_ADC_SamplesReadLatest ( adcHandle, &dataBuffer, 1);

    /*
    Once the previously function returns success, the data is
    in 'dataBuffer'. Application can use this data.
    */
}
```

Code Examples

This topic provides four ADC Driver Library code examples.

Description

Example 1: Auto-sampling, Polled Mode

```
//PIC32 in auto sampling, Polled mode
#define MY_ADC_INSTANCE DRV_ADC_INDEX_0

    DRV_HANDLE              adcHandle;        //Handle returned by DRV_ADC_Open function.
    uint16_t dataBuffer;                      //Buffer to which the data will be written.

    /*
    Open a new client. The handle returned by the this function should be used
    a passing parameter for all the specific client related operations.
    */
    adcHandle = DRV_ADC_Open(MY_ADC_INSTANCE, DRV_IO_INTENT_NONBLOCKING);

    ipHandle =  DRV_ADC_InputsRegister ( adcHandle ,inputSetConfig );

    /*
    ADC is not yet enabled. Enable it once the input set is registered. Starts automatically
    since auto sample mode is enabled.
    */
    DRV_ADC_Start(adcHandle);

    while (DRV_ADC_SampleAvailable(adcHandle))
    {

        DRV_ADC_InputSetRead ( adcHandle, &dataBuffer, 1);

        /*
        Once the previous function returns success, the data is
        in 'dataBuffer'. Application can use this data.
        */
    }
```

Example 2: Manual Triggering, Polled Mode

```
//PIC32 in Manual triggering, Polled mode
#define MY_ADC_INSTANCE DRV_ADC_INDEX_0

int main()
{
    SYS_MODULE_OBJ     myAdcObj;                //Object returned by DRV_ADC_Initialize function.
    DRV_ADC_INIT       adcInitData;             //Contains all the initialization values.
    DRV_HANDLE              adcHandle;          //Handle returned by DRV_ADC_Open function.
    DRV_ADC_INPUTSET_HANDLE  ipHandle;          //Handle returned by DRV_ADC_InputSetRegister function.
    DRV_ADC_INPUTSET_CONFIG     inputSetConfig; //Contains all values to configure the input set.
    uint16_t dataBuffer;                        //Buffer to which the data will be written.

    /*
    These initialization values should be based on the requirement of the
    application and the way it wants the hardware to be operating. Most of
    these values will be written directly to the hardware registers.
    */
    adcInitData.plibModuleId = ADC_ID_1;
    adcInitData.acquisitionTime = PLIB_ADC_ACQUISITION_TIME_15_TAD;
    adcInitData.voltageReference = PLIB_ADC_VREF_POS_TO_VDD_VREF_NEG_TO_VSS;
    adcInitData.clockFrequency = 4000000; //4MHz
    adcInitData.conversionClock = PLIB_ADC_CONV_CLOCK_5_TCY;
    adcInitData.conversionClockSource = PLIB_ADC_CLOCK_SRC_INTERNAL_RC;
    adcInitData.conversionTriggerSource = PLIB_ADC_CONVERSION_TRIGGER_INTERNAL_COUNT;
    adcInitData.dataOutputFormat = PLIB_ADC_OUTPUT_FORMAT_INTEGER_16BIT;

    adcInitData.interruptSource= PLIB_INT_SOURCE_ADC_1;
    adcInitData.samplesPerInterrupt = PLIB_ADC_SAMPLE_PER_INTERRUPT_AT_EACH_SAMPLE;

    myAdcObj = DRV_ADC_Initialize(MY_ADC_INSTANCE, (SYS_MODULE_INIT*)&adcInitData);
```

```
    /*
    Open a new client. The handle returned by the this function should be used
    a passing parameter for all the specific client related operations.
    */
    adcHandle = DRV_ADC_Open(MY_ADC_INSTANCE, DRV_IO_INTENT_NONBLOCKING);

    /*
    Driver invokes the registered callback function every successful data read.
    Application can perform an action or do a state change in the callback
    */
    inputSetConfig.callback = adcCallback;
    inputSetConfig.input    = POTENTIOMETER_ANALOG_INPUT;
    inputSetConfig.errorTolerance = 10;
    inputSetConfig.samplingFrequency = 40000;

    ipHandle =  DRV_ADC_InputSetRegister ( adcHandle ,inputSetConfig );

    /*
    ADC is not yet enabled. Enable it once the inputset is registered. Starts automatically
    since auto sample mode is enabled.
    */
    DRV_ADC_Start(adcHandle);

    while (1)
    {
        /*
        This function can be either called in a sequential way or it could
        be called from a timer periodically
        */
        triggerAdc();
DRV_HANDLE handle;

        DRV_ADC_Tasks (myAdcObj);

        DRV_ADC_InputSetRead ( adcHandle, &dataBuffer, 1);

        /*
        Once the previous function returns success, the data is
        in 'dataBuffer'. Application can use this data.
        */
    }

}

void adcCallback (void)
{
  // Application can do something here
}

void triggerAdc(void)
{
    DRV_ADC_OperationSetup(handle, DRV_ADC_START_SAMPLING);

    /* Give some delay between the two operations. */
    for (i=0; i<100; i++);
    DRV_ADC_OperationSetup(handle, DRV_ADC_START_CONVERSION);

}
```

Example 3: Auto-sampling, Interrupt Mode

```
//PIC32 in auto sampling, Interrupt mode
#define MY_ADC_INSTANCE DRV_ADC_INDEX_0

int main()
{
    SYS_MODULE_OBJ      myAdcObj;            //Object returned by DRV_ADC_Initialize function.
    DRV_ADC_INIT        adcInitData;            //Contains all the initialization values.
    DRV_HANDLE                adcHandle;        //Handle returned by DRV_ADC_Open function.
```

```
    DRV_ADC_INPUTSET_HANDLE  ipHandle;        //Handle returned by DRV_ADC_InputSetRegister function.
    DRV_ADC_INPUTSET_CONFIG     inputSetConfig;//Contains all values to configure the inputset.
    uint16_t dataBuffer;                     //Buffer to which the data will be written.

    /*
    These initialization values should be based on the requirement of the
    application and the way it wants the hardware to be operating. Most of
    these values will be written directly to the hardware registers.
    */
    adcInitData.plibModuleId = ADC_ID_1;
    adcInitData.acquisitionTime = PLIB_ADC_ACQUISITION_TIME_15_TAD;
    adcInitData.voltageReference = PLIB_ADC_VREF_POS_TO_VDD_VREF_NEG_TO_VSS;
    adcInitData.clockFrequency = 4000000; //4MHz
    adcInitData.conversionClock = PLIB_ADC_CONV_CLOCK_5_TCY;
    adcInitData.conversionClockSource = PLIB_ADC_CLOCK_SRC_INTERNAL_RC;
    adcInitData.conversionTriggerSource = PLIB_ADC_CONVERSION_TRIGGER_INTERNAL_COUNT;
    adcInitData.dataOutputFormat = PLIB_ADC_OUTPUT_FORMAT_INTEGER_16BIT;
    adcInitData.initFlags = DRV_ADC_AUTO_SAMPLING;
    adcInitData.interruptSource= PLIB_INT_SOURCE_ADC_1;
    adcInitData.samplesPerInterrupt = PLIB_ADC_SAMPLE_PER_INTERRUPT_AT_EACH_SAMPLE;

    myAdcObj = DRV_ADC_Initialize(MY_ADC_INSTANCE, (SYS_MODULE_INIT*)&adcInitData);

    /*
    Open a new client. The handle returned by the this function should be used
    a passing parameter for all the specific client related operations.
    */
    adcHandle = DRV_ADC_Open(MY_ADC_INSTANCE, DRV_IO_INTENT_NONBLOCKING);

    /*
    Driver invokes the registered callback function every successful data read.
    Application can perform an action or do a state change in the callback
    */
    inputSetConfig.callback = adcCallback;
    inputSetConfig.input    = POTENTIOMETER_ANALOG_INPUT;
    inputSetConfig.errorTolerance = 10;
    inputSetConfig.samplingFrequency = 40000;

    ipHandle =  DRV_ADC_InputSetRegister ( adcHandle ,inputSetConfig );

    /*
    ADC is not yet enabled. Enable it once the inputset is registered. Starts automatically
    since auto sample mode is enabled.
    */
    DRV_ADC_Start(adcHandle);

    while (1)
    {

        /*
        Task function need not be called. Task function
        is registered as ISR in case of interrupt mode
        */

        DRV_ADC_InputSetRead ( adcHandle, &dataBuffer, 1);

        /*
        Once the previous function returns success, the data is
        in 'dataBuffer'. Application can use this data.
        */
    }

}

void adcCallback (void)
{
  // Application can do something here
}
```

```
    // Do something else...

} while(total < MY_BUFFER_SIZE);
```

Example 4: Manual Triggering, Interrupt Mode

```
//PIC32 in Manual triggering, Interrupt mode
#define MY_ADC_INSTANCE DRV_ADC_INDEX_0

int main()
{
    SYS_MODULE_OBJ     myAdcObj;            //Object returned by DRV_ADC_Initialize function.
    DRV_ADC_INIT       adcInitData;            //Contains all the initialization values.
    DRV_HANDLE               adcHandle;        //Handle returned by DRV_ADC_Open function.
    DRV_ADC_INPUTSET_HANDLE  ipHandle;        //Handle returned by DRV_ADC_InputSetRegister function.
    DRV_ADC_INPUTSET_CONFIG     inputSetConfig;//Contains all values to configure the inputset.
    uint16_t dataBuffer;                    //Buffer to which the data will be written.

    /*
    These initialization values should be based on the requirement of the
    application and the way it wants the hardware to be operating. Most of
    these values will be written directly to the hardware registers.
    */
    adcInitData.plibModuleId = ADC_ID_1;
    adcInitData.acquisitionTime = PLIB_ADC_ACQUISITION_TIME_15_TAD;
    adcInitData.voltageReference = PLIB_ADC_VREF_POS_TO_VDD_VREF_NEG_TO_VSS;
    adcInitData.clockFrequency = 4000000; //4MHz
    adcInitData.conversionClock = PLIB_ADC_CONV_CLOCK_5_TCY;
    adcInitData.conversionClockSource = PLIB_ADC_CLOCK_SRC_INTERNAL_RC;
    adcInitData.conversionTriggerSource = PLIB_ADC_CONVERSION_TRIGGER_INTERNAL_COUNT;
    adcInitData.dataOutputFormat = PLIB_ADC_OUTPUT_FORMAT_INTEGER_16BIT;

    adcInitData.interruptSource= PLIB_INT_SOURCE_ADC_1;
    adcInitData.samplesPerInterrupt = PLIB_ADC_SAMPLE_PER_INTERRUPT_AT_EACH_SAMPLE;

    myAdcObj = DRV_ADC_Initialize(MY_ADC_INSTANCE, (SYS_MODULE_INIT*)&adcInitData);

    /*
    Open a new client. The handle returned by the this function should be used
    a passing parameter for all the specific client related operations.
    */
    adcHandle = DRV_ADC_Open(MY_ADC_INSTANCE, DRV_IO_INTENT_NONBLOCKING);

    /*
    Driver invokes the registered callback function every successful data read.
    Application can perform an action or do a state change in the callback
    */
    inputSetConfig.callback = adcCallback;
    inputSetConfig.input    = POTENTIOMETER_ANALOG_INPUT;
    inputSetConfig.errorTolerance = 10;
    inputSetConfig.samplingFrequency = 40000;

    ipHandle =  DRV_ADC_InputSetRegister ( adcHandle ,inputSetConfig );

    /*
    ADC is not yet enabled. Enable it once the inputset is registered. Starts automatically
    since auto sample mode is enabled.
    */
    DRV_ADC_Start(adcHandle);

    while (1)
    {
        /*
        This function can be either called in a sequential way or it could
        be called from a timer periodically
        */
        triggerAdc();

        /*
        Task function need not be called. Task function
```

```
        is registered as ISR in case of interrupt mode
        */

        DRV_ADC_InputSetRead ( adcHandle, &dataBuffer, 1);

        /*
        Once the previous function returns success, the data is
        in 'dataBuffer'. Application can use this data.
        */
    }

}

void adcCallback (void)
{
  // Application can do something here
}

void triggerAdc(void)
{
    DRV_ADC_OperationSetup(handle, DRV_ADC_START_SAMPLING);

    /* Give some delay between the two operations. */
    for (i=0; i<100; i++);
    DRV_ADC_OperationSetup(handle, DRV_ADC_START_CONVERSION);

}
```

Configuring the Library

Macros

	Name	Description
	DRV_ADC_ACQUISITION_TIME	Defines the acquisition time.
	DRV_ADC_ALTERNATE_INPUT_SAMPLING_ENABLE	Enable the alternate input sampling feature of the ADC.
	DRV_ADC_ANALOG_INPUT	Defines the analog input channel.
	DRV_ADC_AUTO_SAMPLING_ENABLE	Enables the auto-sampling feature of the ADC.
	DRV_ADC_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ADC_CONVERSION_CLOCK_PRESCALER	Defines the conversion clock.
	DRV_ADC_CONVERSION_CLOCK_SOURCE	Defines the conversion clock source.
	DRV_ADC_CONVERSION_TRIGGER_SOURCE	Defines the conversion trigger source.
	DRV_ADC_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_ADC_INTERNAL_BUFFER_SIZE	Define the internal buffer size.
	DRV_ADC_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_ADC_INTERRUPT_SOURCE	Defines the interrupt source of the static driver.
	DRV_ADC_PERIPHERAL_ID	ADC PLIB ID Selection
	DRV_ADC_POWER_STATE	Controls the power state of the ADC.
	DRV_ADC_RESULT_FORMAT	Defines the data output format.
	DRV_ADC_SAMPLES_PER_INTERRUPT	Define the sample per interrupt.
	DRV_ADC_STOP_ON_CONVERSION_ENABLE	Enable the stop on conversion feature of the ADC.
	DRV_ADC_VOLTAGE_REFERENCE	Defines the voltage reference.
	DRV_ADC_INDEX	ADC static index selection.
	_DRV_ADC_CONFIG_TEMPLATE_H	This is macro _DRV_ADC_CONFIG_TEMPLATE_H.

Description

The configuration of the ADC device driver is based on the file `system_config.h`.

This header file contains the configuration selection for the ADC device driver build. Based on the selections made here and the system setup, the ADC device driver may support the selected features. These configuration settings will apply to all instances of the device driver.

This header can be placed anywhere in the application specific folders and the path of this header needs to be presented to the include search for a successful build. Refer to the Applications Overview section for more details.

DRV_ADC_ACQUISITION_TIME Macro

Defines the acquisition time.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_ACQUISITION_TIME ADC_ACQUISITION_TIME_4_TAD
```

Description

ADC Acquisition Time

This macro defines the acquisition time of the ADC driver. This provides static override of the dynamic selection of the acquisition time. If this macro is defined, this will be used for setting up the acquisition time and not the acquisition time value provided by DRV_ADC_INIT.

Remarks

None.

DRV_ADC_ALTERNATE_INPUT_SAMPLING_ENABLE Macro

Enable the alternate input sampling feature of the ADC.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_ALTERNATE_INPUT_SAMPLING_ENABLE false
```

Description

ADC Alternate Input Sampling Enable

This macro enables the alternate input sampling feature of the ADC. This macro can take the following values:

- true - Enables the alternate Input sampling feature of the ADC
- false - Disables the alternate Input sampling feature of the ADC
- DRV_CONFIG_NOT_SUPPORTED - When the feature is not supported on the instance

Remarks

None.

DRV_ADC_ANALOG_INPUT Macro

Defines the analog input channel.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_ANALOG_INPUT ADC_INPUT_AN2
```

Description

ADC Analog input channel

This macro defines the analog input channel for the ADC driver. This provides static override of the dynamic selection of the analog input. If this macro is defined, this will be used for setting up the analog input and not the analog input value provided by DRV_ADC_INIT.

Remarks

None.

DRV_ADC_AUTO_SAMPLING_ENABLE Macro

Enables the auto-sampling feature of the ADC.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_AUTO_SAMPLING_ENABLE true
```

Description

ADC Auto Sampling Enable

This macro enables the auto-sampling feature of the ADC. This macro can take the following values:

- true - Enables the auto-sampling feature of the ADC
- false - Disables the auto-sampling feature of the ADC
- DRV_CONFIG_NOT_SUPPORTED - When the feature is not supported on the instance

Remarks

None.

DRV_ADC_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_CLIENTS_NUMBER 1
```

Description

ADC Maximum Number of Clients

This definition selected the maximum number of clients that the ADC driver can support at run time.

Remarks

None.

DRV_ADC_CONVERSION_CLOCK_PRESCALER Macro

Defines the conversion clock.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_CONVERSION_CLOCK_PRESCALER ADC_CONV_CLOCK_4_TCY
```

Description

ADC Conversion Clock

This macro defines the conversion clock for the ADC driver. This provides static override of the dynamic selection of

the conversion clock. If this macro is defined, this will be used for setting up the conversion clock and not the conversion clock value provided by DRV_ADC_INIT.

Remarks

None.

DRV_ADC_CONVERSION_CLOCK_SOURCE Macro

Defines the conversion clock source.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_CONVERSION_CLOCK_SOURCE ADC_CLOCK_SRC_SYSTEM_CLOCK
```

Description

ADC Conversion Clock Source

This macro defines the conversion clock source for the ADC driver. This provides static override of the dynamic selection of the conversion clock source. If this macro is defined, this will be used for setting up the conversion clock source and not the conversion clock source value provided by DRV_ADC_INIT.

Remarks

None.

DRV_ADC_CONVERSION_TRIGGER_SOURCE Macro

Defines the conversion trigger source.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_CONVERSION_TRIGGER_SOURCE ADC_CONVERSION_TRIGGER_INTERNAL_COUNT
```

Description

Conversion Trigger Source

This macro defines the conversion trigger source for the ADC driver. This provides static override of the dynamic selection of the conversion trigger source. If this macro is defined, this will be used for setting up the conversion trigger source and not the conversion trigger source value provided by DRV_ADC_INIT.

Remarks

None.

DRV_ADC_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_INSTANCES_NUMBER 1
```

Description

ADC hardware instance configuration

This macro sets up the maximum number of hardware instances that can be supported.

Remarks

None.

DRV_ADC_INTERNAL_BUFFER_SIZE Macro

Define the internal buffer size.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_INTERNAL_BUFFER_SIZE 2
```

Description

ADC Internal buffer size

This macro defines the internal buffer size.

Remarks

None.

DRV_ADC_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_INTERRUPT_MODE true
```

Description

ADC Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of ADC operation is desired
- false - Select if polling mode of ADC operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

DRV_ADC_INTERRUPT_SOURCE Macro

Defines the interrupt source of the static driver.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_INTERRUPT_SOURCE PLIB_INT_SOURCE_ADC_1
```

Description

ADC Interrupt Source

Macro to define the interrupt source of the static driver.

Remarks

Refer to the Interrupt Peripheral Library document for more information on the PLIB_INT_SOURCE enumeration.

DRV_ADC_PERIPHERAL_ID Macro

ADC PLIB ID Selection

File

drv_adc_config_template.h

C

```
#define DRV_ADC_PERIPHERAL_ID ADC_ID_1
```

Description

ADC PLIB ID Selection

This macro selects the ADC PLIB ID Selection. This is an initialization override of the adcID member of the initialization configuration.

Remarks

None.

DRV_ADC_POWER_STATE Macro

Controls the power state of the ADC.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_POWER_STATE SYS_MODULE_POWER_IDLE_STOP
```

Description

ADC power state configuration

This macro controls the power state of the ADC.

Remarks

This feature may not be available in the device or the ADC module selected.

DRV_ADC_RESULT_FORMAT Macro

Defines the data output format.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_RESULT_FORMAT ADC_RESULT_FORMAT_INTEGER_16BIT
```

Description

ADC Data Output Format

This macro defines the data output format for the ADC driver. This provides static override of the dynamic selection of the data output format. If this macro is defined, this will be used for setting up the data output format and not the data output format value provided by DRV_ADC_INIT.

Remarks

None.

DRV_ADC_SAMPLES_PER_INTERRUPT Macro

Define the sample per interrupt.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_SAMPLES_PER_INTERRUPT 2
```

Description

Samples per Interrupt

This macro defines the samples per interrupt of the ADC driver. This provides static override of the dynamic selection of the sample per interrupt. If this macro is defined, this will be used for setting up the samples per interrupt and not the samples per interrupt value provided by DRV_ADC_INIT.

Remarks

Select this size based on the device available and the number of samples that are required to form a set.

DRV_ADC_STOP_ON_CONVERSION_ENABLE Macro

Enable the stop on conversion feature of the ADC.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_STOP_ON_CONVERSION_ENABLE false
```

Description

ADC Stop on conversion Enable

This macro enables the stop on conversion feature of the ADC. This macro can take the following values:

- true - Enables the ADC to stop on conversion
- false - Disables the ADC to stop on conversion
- DRV_CONFIG_NOT_SUPPORTED - When the feature is not supported on the instance

Remarks

None.

DRV_ADC_VOLTAGE_REFERENCE Macro

Defines the voltage reference.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_VOLTAGE_REFERENCE ADC_VREF_POS_TO_VDD_VREF_NEG_TO_VSS
```

Description

ADC Voltage Reference

This macro defines the voltage reference of the ADC driver. This provides static override of the dynamic selection of the voltage reference. If this macro is defined, this will be used for setting up the voltage reference and not the voltage reference value provided by DRV_ADC_INIT.

Remarks

None.

DRV_ADC_INDEX Macro

ADC static index selection.

File

drv_adc_config_template.h

C

```
#define DRV_ADC_INDEX DRV_ADC_INDEX_0
```

Description

ADC Static Index Selection

ADC static index selection for the driver object reference.

Remarks

This index is required to make a reference to the driver object.

_DRV_ADC_CONFIG_TEMPLATE_H Macro

File

drv_adc_config_template.h

C

```
#define _DRV_ADC_CONFIG_TEMPLATE_H
```

Description

This is macro _DRV_ADC_CONFIG_TEMPLATE_H.

Building the Library

This section lists the files that are available in the SD Card Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/adc`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_adc.h	This file provides the interface definitions of the ADC driver.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_adc_hw_dynamic.c	This file contains the core implementation of the ADC driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The ADC Driver Library depends on the following modules:

- Clock System Service Library
- Interrupt System Service Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_ADC_Deinitialize	Deinitializes the specified instance of the ADC driver module. **Implementation:** Static/Dynamic
	DRV_ADC_Initialize	Initializes the ADC driver. **Implementation:** Static/Dynamic
	DRV_ADC_Status	Provides the current status of the ADC driver module. **Implementation:** Dynamic
	DRV_ADC_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic

b) Client Core Configuration Functions

	Name	Description
	DRV_ADC_ClientStatus	Gets the current client-specific status the ADC driver. **Implementation:** Dynamic
	DRV_ADC_Close	Closes an opened-instance of the ADC driver. **Implementation:** Static/Dynamic
	DRV_ADC_InputsRegister	Registers an input set with the driver for sampling. **Implementation:** Dynamic
	DRV_ADC_Open	Opens the specified ADC driver instance and returns a handle to it. **Implementation:** Static/Dynamic
	DRV_ADC_SamplesAvailable	Identifies if any the ADC driver has any samples available to read. **Implementation:** Static/Dynamic
	DRV_ADC_Start	Starts the ADC driver sampling and converting analog to digital values. **Implementation:** Static/Dynamic
	DRV_ADC_Stop	Stops the ADC driver from sampling and converting analog to digital values. **Implementation:** Static/Dynamic
	DRV_ADC_ChannelExtendedScanInputsAdd	Adds extended scan input on the supported devices. **Implementation:** Static
	DRV_ADC_ChannelExtendedScanInputsRemove	Removes extended scan input on the supported devices. **Implementation:** Static
	DRV_ADC_ChannelScanInputsAdd	Adds scan input. **Implementation:** Static
	DRV_ADC_ChannelScanInputsRemove	Removes scan input. **Implementation:** Static
	DRV_ADC_NegativeInputSelect	Selects the negative input. **Implementation:** Static
	DRV_ADC_PositiveInputSelect	Selects the positive input. **Implementation:** Static

© 2013-2015 Microchip Technology Inc.

c) Other Functions

	Name	Description
	DRV_ADC_SamplesRead	Reads the converted sample data from the ADC driver. **Implementation:** Static/Dynamic
	DRV_ADC_SamplesReadLatest	Reads the most recently converted sample data from the ADC driver. **Implementation:** Dynamic

d) Data Types and Constants

	Name	Description
	DRV_ADC_CLIENT_STATUS	Defines the client-specific status of the ADC driver.
	DRV_ADC_INIT	Defines the data required to initialize or reinitialize the ADC driver.
	DRV_ADC_INIT_FLAGS	Identifies the initialization flags of the ADC module.
	DRV_ADC_INDEX_0	ADC driver index definitions.
	DRV_ADC_INDEX_1	This is macro DRV_ADC_INDEX_1.
	DRV_ADC_INDEX_2	This is macro DRV_ADC_INDEX_2.
	DRV_ADC_BUFFER_HANDLE	This type defines the ADC Driver Buffer handle.
	DRV_ADC_BUFFER_STATUS	Specifies the status of the buffer for the read, write and erase operations.
	DRV_ADC_EVENT	Identifies the possible events that can result from a request.
	DRV_ADC_EVENT_HANDLER	Pointer to a ADC Driver Event handler function

Description

This section lists the interface routines, data types, constants and macros for the library.

a) System Interaction Functions

DRV_ADC_Deinitialize Function

Deinitializes the specified instance of the ADC driver module.

Implementation: Static/Dynamic

File

drv_adc.h

C

```
void DRV_ADC_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the specified instance of the ADC driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_ADC_Status operation. The system has to use DRV_ADC_Status to find out when the module is in the ready state.

Preconditions

The DRV_ADC_Initialize function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_ADC_Initialize
SYS_STATUS          status;


DRV_ADC_Deinitialize(object);

status = DRV_ADC_Status(object);
if (SYS_MODULE_DEINITIALIZED == status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_ADC_Initialize

Function

void DRV_ADC_Deinitialize(SYS_MODULE_OBJ object)

DRV_ADC_Initialize Function

Initializes the ADC driver.

Implementation: Static/Dynamic

File

drv_adc.h

C

```
SYS_MODULE_OBJ DRV_ADC_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the ADC driver, making it ready for clients to open and use it.

Remarks

This function must be called before any other ADC function is called.

This function should only be called once during system initialization unless DRV_ADC_Deinitialize is called to deinitialize the driver instance.

This function will NEVER block for hardware access. If the operation requires time to allow the hardware to reinitialize, it will be reported by the DRV_ADC_Status operation. The system must use DRV_ADC_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
DRV_ADC_INIT    init;
SYS_MODULE_OBJ  objectHandle;

// Populate the init structure
init.moduleInit.value           = SYS_MODULE_POWER_RUN_FULL;
init.adcID                      = ADC_ID_1;
init.initFlags                  = DRV_ADC_AUTO_SAMPLING;
init.acquisitionTime            = ADC_ACQUISITION_TIME_15_TAD;
init.voltageReference           = ADC_VREF_POS_TO_VDD_VREF_NEG_TO_VSS;
init.conversionClockPrescaler   = ADC_CONV_CLOCK_5_TCY;
init.conversionClockSource      = ADC_CLOCK_SRC_INTERNAL_RC;
init.conversionTriggerSource    = ADC_CONVERSION_TRIGGER_INTERNAL_COUNT;
init.samplesPerInterrupt        = ADC_SAMPLE_PER_INTERRUPT_AT_EACH_SAMPLE;
init.resultFormat               = ADC_RESULT_FORMAT_INTEGER_16BIT;
init.analogInput                = ADC_INPUT_AN2
init.interruptSource            = INT_SOURCE_ADC_1;

// Initialize the ADC driver
objectHandle = DRV_ADC_Initialize(DRV_ADC_INDEX_0, (SYS_MODULE_INIT*)&init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Index for the driver instance to be initialized

init	Pointer to a data structure containing any data necessary to initialize the driver. This pointer may be null if no data is required because static overrides have been provided.

Function

SYS_MODULE_OBJ DRV_ADC_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

DRV_ADC_Status Function

Provides the current status of the ADC driver module.

Implementation: Dynamic

File

drv_adc.h

C

```
SYS_STATUS DRV_ADC_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another

Description

This function provides the current status of the ADC driver module.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_STATUS_BUSY - Indicates that the driver is busy with a previous system level operation and cannot start another

SYS_STATUS_ERROR - Indicates that the driver is in an error state

Any value less than SYS_STATUS_ERROR is also an error state.

SYS_MODULE_DEINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR

The this operation can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, the a previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_ADC_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_ADC_Initialize
SYS_STATUS          status;

status = DRV_ADC_Status(object);
else if (SYS_STATUS_ERROR >= status)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_ADC_Initialize

Function

SYS_STATUS DRV_ADC_Status (SYS_MODULE_OBJ object)

© 2013-2015 Microchip Technology Inc.

DRV_ADC_Tasks Function

Maintains the driver's state machine and implements its ISR.

Implementation: Dynamic

File

drv_adc.h

C

```
void DRV_ADC_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal state machine and implement its ISR for interrupt-driven implementations.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks function (SYS_Tasks) or by the appropriate raw ISR.

This function may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_ADC_Initialize function must have been called for the specified ADC driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_ADC_Initialize

while (true)
{
    DRV_ADC_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_ADC_Initialize)

Function

void DRV_ADC_Tasks (SYS_MODULE_OBJ object)

b) Client Core Configuration Functions

DRV_ADC_ClientStatus Function

Gets the current client-specific status the ADC driver.

Implementation: Dynamic

File

drv_adc.h

C

```
DRV_ADC_CLIENT_STATUS DRV_ADC_ClientStatus(DRV_HANDLE handle);
```

Returns

A DRV_ADC_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the ADC driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_ADC_Initialize function must have been called.

DRV_ADC_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE              handle;  // Returned from DRV_ADC_Open
DRV_ADC_CLIENT_STATUS   status;

status = DRV_ADC_ClientStatus(handle);
if(DRV_ADC_CLIENT_STATUS_ERROR >= status)
{
    // Handle the error
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open

Function

DRV_ADC_CLIENT_STATUS DRV_ADC_ClientStatus(DRV_HANDLE handle)

DRV_ADC_Close Function

Closes an opened-instance of the ADC driver.

Implementation: Static/Dynamic

File

drv_adc.h

C

```
SYS_STATUS DRV_ADC_Close(DRV_HANDLE handle);
```

Returns

SYS_STATUS - System Status

Description

This function closes an opened-instance of the ADC driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver functions. A new handle must be obtained by calling DRV_ADC_Open before the caller may use the driver again.

If DRV_IO_INTENT_BLOCKING was requested and the driver was built appropriately to support blocking behavior call may block until the operation is complete.

If DRV_IO_INTENT_NON_BLOCKING request the driver client can call the DRV_ADC_Status operation to find out when the module is in the ready state (the handle is no longer valid).

Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_ADC_Initialize function must have been called for the specified ADC driver instance.

DRV_ADC_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_ADC_Open

DRV_ADC_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_STATUS DRV_ADC_Close(DRV_Handle handle)

DRV_ADC_InputsRegister Function

Registers an input set with the driver for sampling.

Implementation: Dynamic

File

drv_adc.h

C

```
void DRV_ADC_InputsRegister(DRV_HANDLE handle, uint32_t inputsMask);
```

Returns

None.

Description

This function registers an input set with the driver for sampling.

Remarks

None.

Preconditions

The DRV_ADC_Initialize function must have been called for the specified ADC device instance and the DRV_ADC_Status must have returned SYS_STATUS_READY.

DRV_ADC_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_ADC_Open

DRV_ADC_InputsRegister (handle, ADC_INPUT_AN2|ADC_INPUT_AN3);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
inputsMask	Mask bits recognizing the various Analog Channels

Function

void DRV_ADC_InputsRegister (DRV_HANDLE handle , uint32_t inputsMask)

DRV_ADC_Open Function

Opens the specified ADC driver instance and returns a handle to it.

Implementation: Static/Dynamic

File

drv_adc.h

C

```
DRV_HANDLE DRV_ADC_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID.

Description

This function opens the specified ADC driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_ADC_Close function is called.

This function will NEVER block waiting for hardware.

If the DRV_IO_INTENT_BLOCKING is requested and the driver was built appropriately to support blocking behavior, other client-level operations may block waiting on hardware until they are complete.

If DRV_IO_INTENT_NON_BLOCKING is requested the driver client can call the DRV_ADC_ClientStatus operation to find out when the module is in the ready state.

If the requested intent flags are not supported, the function will return DRV_HANDLE_INVALID.

Preconditions

The DRV_ADC_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_ADC_Open(DRV_ADC_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_ADC_Open(const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT intent)

DRV_ADC_SamplesAvailable Function

Identifies if any the ADC driver has any samples available to read.

Implementation: Static/Dynamic

File

drv_adc.h

C

```
bool DRV_ADC_SamplesAvailable(DRV_HANDLE handle);
```

Returns

- true - If one or more samples are available for the registered input set
- false - If no samples are available

Description

This function identifies if any the ADC driver has any samples available to read.

Remarks

None.

Preconditions

The DRV_ADC_Initialize function must have been called.

DRV_ADC_Open must have been called to obtain a valid opened device handle.

The desired analog input set must have been selected by calling DRV_ADC_InputsRegister.

DRV_ADC_Start must have been called to start the driver sampling and converting analog input samples to digital values.

Example

```
DRV_HANDLE      handle;  // Returned from DRV_ADC_Open
DRV_ADC_SAMPLE  buffer;

// An input set must have been registered and the ADC started.

if (DRV_ADC_SamplesAvailable(handle))
{
    DRV_ADC_SamplesRead(handle, &buffer, sizeof(buffer));
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_ADC_SamplesAvailable (DRV_HANDLE handle)

DRV_ADC_Start Function

Starts the ADC driver sampling and converting analog to digital values.

Implementation: Static/Dynamic

File

drv_adc.h

C

```
void DRV_ADC_Start(DRV_HANDLE handle);
```

Returns

None.

Description

This function starts the ADC driver sampling the selected analog inputs and converting the samples to digital values.

Remarks

Call DRV_ADC_SamplesAvailable to find out when one or more samples is available.

Preconditions

The DRV_ADC_Initialize function must have been called.

DRV_ADC_Open must have been called to obtain a valid opened device handle.

The desired analog input set must have been selected by calling DRV_ADC_InputsRegister.

Example

```
DRV_HANDLE handle;  // Returned from DRV_ADC_Open

// Use DRV_ADC_InputsRegister to register the desired inputs.

DRV_ADC_Start(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_ADC_Start(DRV_HANDLE handle)

DRV_ADC_Stop Function

Stops the ADC driver from sampling and converting analog to digital values.

Implementation: Static/Dynamic

File

drv_adc.h

C

```
void DRV_ADC_Stop(DRV_HANDLE handle);
```

Returns

None.

Description

This function stops the ADC driver from sampling analog inputs and converting the samples to digital values.

Remarks

Call DRV_ADC_Start to restart sampling and conversion of analog inputs to digital values.

Preconditions

The DRV_ADC_Initialize function must have been called.

DRV_ADC_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_ADC_Open

DRV_ADC_Stop(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_ADC_Stop(DRV_HANDLE handle)

DRV_ADC_ChannelExtendedScanInputsAdd Function

Adds extended scan input on the supported devices.

Implementation: Static

File

drv_adc.h

C

```
inline void DRV_ADC_ChannelExtendedScanInputsAdd(ADC_INPUTS_SCAN_EXTENDED eScanInput);
```

Returns

None.

Description

This function adds extended scan input.

Remarks

None.

Preconditions

The DRV_ADC_Initialize function must have been called.

Example

```
DRV_ADC_ChannelExtendedScanInputsAdd(ADC_INPUT_SCAN_AN36);
```

Parameters

Parameters	Description
scanInput	Extended scan input (AN32...ANx)

Function

inline void DRV_ADC_ChannelExtendedScanInputsAdd

(

ADC_INPUTS_SCAN_EXTENDED eScanInput

)

DRV_ADC_ChannelExtendedScanInputsRemove Function

Removes extended scan input on the supported devices.

Implementation: Static

File

drv_adc.h

C

```
inline void DRV_ADC_ChannelExtendedScanInputsRemove(ADC_INPUTS_SCAN_EXTENDED eScanInput);
```

Returns

None.

Description

This function removes extended scan input.

Remarks

None.

Preconditions

The DRV_ADC_Initialize function must have been called.

Example

```
DRV_ADC_ChannelExtendedScanInputsRemove(ADC_INPUT_SCAN_AN36);
```

Parameters

Parameters	Description
scanInput	Extended scan input (AN32...ANx)

Function

inline void DRV_ADC_ChannelExtendedScanInputsRemove

(

ADC_INPUTS_SCAN_EXTENDED eScanInput

)

DRV_ADC_ChannelScanInputsAdd Function

Adds scan input.

Implementation: Static

File

drv_adc.h

C

```
inline void DRV_ADC_ChannelScanInputsAdd(ADC_INPUTS_SCAN scanInput);
```

Returns

None.

Description

This function adds scan input.

Remarks

None.

Preconditions

The DRV_ADC_Initialize function must have been called.

Example

```
DRV_ADC_ChannelScanInputsAdd(ADC_INPUT_SCAN_AN5);
```

Parameters

Parameters	Description
scanInput	Scan input (AN0...ANx)

Function

inline void DRV_ADC_ChannelScanInputsAdd(ADC_INPUTS_SCAN scanInput)

DRV_ADC_ChannelScanInputsRemove Function

Removes scan input.

Implementation: Static

File

drv_adc.h

C

```
inline void DRV_ADC_ChannelScanInputsRemove(ADC_INPUTS_SCAN scanInput);
```

Returns

None.

Description

This function removes scan input.

Remarks

None.

Preconditions

The DRV_ADC_Initialize function must have been called.

Example

```
DRV_ADC_ChannelScanInputsRemove(ADC_INPUT_SCAN_AN5);
```

Parameters

Parameters	Description
scanInput	Scan input (AN0...ANx)

Function

inline void DRV_ADC_ChannelScanInputsRemove(ADC_INPUTS_SCAN scanInput)

DRV_ADC_NegativeInputSelect Function

Selects the negative input.

Implementation: Static

File

drv_adc.h

C

```
inline void DRV_ADC_NegativeInputSelect(ADC_MUX mux, ADC_INPUTS_NEGATIVE input);
```

Returns

None.

Description

This function selects the negative input .

Remarks

None.

Preconditions

The DRV_ADC_Initialize function must have been called.

Example

```
DRV_ADC_NegativeInputSelect(ADC_MUX_A, ADC_INPUT_NEGATIVE_VREF_MINUS);
```

Parameters

Parameters	Description
mux	Input mux (MUXA/MUXB) for which the negative input is being selected

Function

inline void DRV_ADC_NegativeInputSelect(

ADC_MUX mux,

ADC_INPUTS_NEGATIVE input

)

DRV_ADC_PositiveInputSelect Function

Selects the positive input.

Implementation: Static

File

drv_adc.h

C

```
inline void DRV_ADC_PositiveInputSelect(ADC_MUX mux, ADC_INPUTS_POSITIVE input);
```

Returns

None.

Description

This function selects the positive input .

Remarks

None.

Preconditions

The DRV_ADC_Initialize function must have been called.

Example

```
DRV_ADC_PositiveInputSelect(ADC_MUX_A, ADC_INPUT_POSITIVE_AN2);
```

Parameters

Parameters	Description
mux	Input mux (MUXA/MUXB) for which the negative input is being selected

Function

inline void DRV_ADC_PositiveInputSelect(

ADC_MUX mux,

ADC_INPUTS_POSITIVE input

)

c) Other Functions

DRV_ADC_SamplesRead Function

Reads the converted sample data from the ADC driver.

Implementation: Static/Dynamic

File

drv_adc.h

C

```
unsigned short DRV_ADC_SamplesRead(DRV_HANDLE handle, ADC_SAMPLE * buffer, unsigned short
bufferSize);
```

Returns

Number of bytes of sample data copied to the specified buffer.

Description

This function reads converted sample data from the ADC driver into the given buffer. How many samples depends on how many samples are available and on the relative sizes of the samples and the buffer passed in.

Zero (0) samples are copied if the bufferSize is less than the size of a complete set of samples for the registered inputs.

N sets of samples where the bufferSize / size of a complete set of samples = N, unless less than N samples are currently available. Then, the number of samples currently available are copied.

Remarks

The DRV_ADC_SamplesAvailable function can be used to determine if any sample data is available.

Calling this function removes the samples from the driver's internal buffer queue of samples.

Preconditions

The DRV_ADC_Initialize function must have been called.

DRV_ADC_Open must have been called to obtain a valid opened device handle.

The desired analog input set must have been selected by calling DRV_ADC_InputsRegister.

DRV_ADC_Start must have been called to start the driver sampling and converting analog input samples to digital values.

Example

```
DRV_HANDLE      handle;  // Returned from DRV_ADC_Open
ADC_SAMPLE      buffer;

// An input set must have been registered and the ADC started.

if (DRV_ADC_SamplesAvailable(handle))
{
    DRV_ADC_SamplesRead(handle, &buffer, sizeof(buffer));
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
buffer	A pointer to the buffer to where the sample data will be copied
bufferSize	Size of the buffer (in bytes)

Function

unsigned short DRV_ADC_SamplesRead (DRV_HANDLE handle,

ADC_SAMPLE *buffer,

unsigned short bufferSize);

DRV_ADC_SamplesReadLatest Function

Reads the most recently converted sample data from the ADC driver.

Implementation: Dynamic

File

drv_adc.h

C

```
unsigned short DRV_ADC_SamplesReadLatest(DRV_HANDLE handle, ADC_SAMPLE * buffer, unsigned short
bufferSize);
```

Returns

Number of bytes of sample data copied to the specified buffer.

Description

This function reads only the most recently converted sample data from the ADC driver into the given buffer. Only the data for a single set of samples for the registered inputs is copied to the caller's buffer. If the buffer size is less than the size of a complete set of samples for the registered inputs, no data is copied to the caller's buffer. Also, no sample data is copied to the caller's buffer if no sample data is currently available.

Remarks

The DRV_ADC_SamplesAvailable function can be used to determine if any sample data is available.

This function does not remove any data from the driver's internal buffer queue of sample data.

Preconditions

The DRV_ADC_Initialize function must have been called.

DRV_ADC_Open must have been called to obtain a valid opened device handle.

The desired analog input set must have been selected by calling DRV_ADC_InputsRegister.

DRV_ADC_Start must have been called to start the driver sampling and converting analog input samples to digital values.

Example

```
DRV_HANDLE      handle;  // Returned from DRV_ADC_Open
ADC_SAMPLE      buffer;

// An input set must have been registered and the ADC started.

if (DRV_ADC_SamplesAvailable(handle))
{
    DRV_ADC_SamplesReadLatest(handle, &buffer, sizeof(buffer));
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
buffer	A pointer to the buffer to where the sample data will be copied
bufferSize	Size of the buffer (in bytes)

Function

unsigned short DRV_ADC_SamplesReadLatest (DRV_HANDLE handle,

ADC_SAMPLE *buffer,

unsigned short bufferSize)

d) Data Types and Constants

DRV_ADC_CLIENT_STATUS Enumeration

Defines the client-specific status of the ADC driver.

File

drv_adc.h

C

```
typedef enum {
  DRV_ADC_CLIENT_STATUS_STARTED,
  DRV_ADC_CLIENT_STATUS_STOPPED,
  DRV_ADC_CLIENT_STATUS_READY,
  DRV_ADC_CLIENT_STATUS_BUSY,
  DRV_ADC_CLIENT_STATUS_INVALID,
  DRV_ADC_CLIENT_STATUS_OVERFLOW,
  DRV_ADC_CLIENT_STATUS_BUFFER_TOO_SMALL
} DRV_ADC_CLIENT_STATUS;
```

Members

Members	Description
DRV_ADC_CLIENT_STATUS_STARTED	ADC Started
DRV_ADC_CLIENT_STATUS_STOPPED	stopped on error
DRV_ADC_CLIENT_STATUS_READY	Driver OK, ready for client operations
DRV_ADC_CLIENT_STATUS_BUSY	An operation is currently in progress
DRV_ADC_CLIENT_STATUS_INVALID	Client in an invalid (or unopened) state
DRV_ADC_CLIENT_STATUS_OVERFLOW	Driver Overflowed
DRV_ADC_CLIENT_STATUS_BUFFER_TOO_SMALL	Input Set Read Buffer Too Small

Description

ADC Client Status

This enumeration defines the client-specific status codes of the ADC driver.

Remarks

Returned by the DRV_ADC_ClientStatus function.

DRV_ADC_INIT Structure

Defines the data required to initialize or reinitialize the ADC driver.

File

drv_adc.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  ADC_MODULE_ID adcId;
  ADC_CLOCK_SOURCE adcClkSrc;
  ADC_CONVERSION_CLOCK tadClk;
  ADC_ACQUISITION_TIME acquisitionTime;
  ADC_CONVERSION_TRIGGER_SOURCE convTrgSrc;
  DRV_ADC_INIT_FLAGS initFlags;
  ADC_VOLTAGE_REFERENCE voltageReference;
  ADC_SAMPLES_PER_INTERRUPT samplesPerInterrupt;
  ADC_RESULT_FORMAT resultFormat;
  ADC_INPUTS_POSITIVE analogInput;
  INT_SOURCE interruptSource;
} DRV_ADC_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
ADC_MODULE_ID adcId;	Identifies ADC hardware module (PLIB-level) ID
ADC_CLOCK_SOURCE adcClkSrc;	Selects input clock source
ADC_CONVERSION_CLOCK tadClk;	Conversion clock value
ADC_ACQUISITION_TIME acquisitionTime;	Data acquisition time in auto sampling mode
ADC_CONVERSION_TRIGGER_SOURCE convTrgSrc;	Conversion Trigger Source
DRV_ADC_INIT_FLAGS initFlags;	Initialization Flags
ADC_VOLTAGE_REFERENCE voltageReference;	Voltage Reference Selection
ADC_SAMPLES_PER_INTERRUPT samplesPerInterrupt;	Samples Per Interrupt valid values = 0- 15
ADC_RESULT_FORMAT resultFormat;	Result Format
ADC_INPUTS_POSITIVE analogInput;	Input Channel to convert
INT_SOURCE interruptSource;	Interrupt Source for the module

Description

ADC Driver Initialization Data

This structure defines the data required to initialize or reinitialize the ADC driver.

Remarks

Not all init features are available for all devices. Refer to the specific data sheet to determine availability.

DRV_ADC_INIT_FLAGS Enumeration

Identifies the initialization flags of the ADC module.

File

drv_adc.h

C

```
typedef enum {
  DRV_ADC_STOP_CONVERSION_ON_INTERRUPT,
  DRV_ADC_ALTERNATE_INPUT_SAMPLING,
  DRV_ADC_AUTO_SAMPLING,
  DRV_ADC_MANUAL_SAMPLING
} DRV_ADC_INIT_FLAGS;
```

Members

Members	Description
DRV_ADC_STOP_CONVERSION_ON_INTERRUPT	Stops the conversion on the interrupt
DRV_ADC_ALTERNATE_INPUT_SAMPLING	Alternate Input Sampling
DRV_ADC_AUTO_SAMPLING	Begin sampling automatically after previous conversion
DRV_ADC_MANUAL_SAMPLING	Manual Sampling

Description

ADC Initialization Flags

This data type identifies the initialization flags of the ADC module.

Remarks

Not all modes are available on all devices. Refer to the specific data sheet to determine availability.

DRV_ADC_INDEX_0 Macro

ADC driver index definitions.

File

drv_adc.h

C

```
#define DRV_ADC_INDEX_0 0
```

Description

Driver ADC Module Index Numbers

These constants provide ADC driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_ADC_Initialize function to identify the driver instance in use.

DRV_ADC_INDEX_1 Macro

File

drv_adc.h

C

```
#define DRV_ADC_INDEX_1 1
```

Description

This is macro DRV_ADC_INDEX_1.

DRV_ADC_INDEX_2 Macro

File

drv_adc.h

C

```
#define DRV_ADC_INDEX_2 2
```

Description

This is macro DRV_ADC_INDEX_2.

DRV_ADC_BUFFER_HANDLE Type

This type defines the ADC Driver Buffer handle.

File

drv_adc.h

C

```
typedef uintptr_t DRV_ADC_BUFFER_HANDLE;
```

Description

ADC Driver Buffer Handle

This type defines the ADC Driver Buffer handle.

Remarks

None.

DRV_ADC_BUFFER_STATUS Enumeration

Specifies the status of the buffer for the read, write and erase operations.

File

drv_adc.h

C

```
typedef enum {
  DRV_ADC_BUFFER_COMPLETED = 0,
  DRV_ADC_BUFFER_QUEUED = 1,
  DRV_ADC_BUFFER_IN_PROGRESS = 2,
  DRV_ADC_BUFFER_ERROR_UNKNOWN = -1
} DRV_ADC_BUFFER_STATUS;
```

Members

Members	Description
DRV_ADC_BUFFER_COMPLETED = 0	Done OK and ready
DRV_ADC_BUFFER_QUEUED = 1	Scheduled but not started
DRV_ADC_BUFFER_IN_PROGRESS = 2	Currently being in transfer
DRV_ADC_BUFFER_ERROR_UNKNOWN = -1	Unknown buffer

Description

ADC Driver Buffer Status

ADC Driver Buffer Status

This type specifies the status of the buffer for the read, write and erase operations.

Remarks

None.

DRV_ADC_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_adc.h

C

```
typedef enum {
  DRV_ADC_EVENT_BUFFER_COMPLETE,
  DRV_ADC_EVENT_BUFFER_ERROR
} DRV_ADC_EVENT;
```

Members

Members	Description
DRV_ADC_EVENT_BUFFER_COMPLETE	Buffer operation has been completed successfully. Read/Write/Erase Complete
DRV_ADC_EVENT_BUFFER_ERROR	There was an error during the block operation Read/Write/Erase Error

Description

ADC Driver Events

This enumeration identifies the possible events that can result from a Write, or Erase request caused by the client. Note that the ADC driver does not generate events for read.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_ADC_EventHandlerSet function when a block request is completed.

DRV_ADC_EVENT_HANDLER Type

Pointer to a ADC Driver Event handler function

File

drv_adc.h

C

```
typedef void (* DRV_ADC_EVENT_HANDLER)(DRV_ADC_EVENT event, DRV_ADC_BUFFER_HANDLE bufferHandle,
uintptr_t context);
```

Returns

None.

Description

ADC Driver Event Handler Function Pointer

This data type defines the required function signature for the ADC event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values and return value are described here and a partial example implementation is provided.

Remarks

If the event is DRV_ADC_EVENT_BUFFER_COMPLETE, it means that the write or a erase operation was completed successfully.

If the event is DRV_ADC_EVENT_BUFFER_ERROR, it means that the scheduled operation was not completed successfully.

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_ADC_EventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_MyBufferEventHandler
(
    DRV_ADC_EVENT event,
    DRV_ADC_BUFFER_HANDLE bufferHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_ADC_EVENT_BUFFER_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_ADC_EVENT_BUFFER_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write/Erase requests
context	Value identifying the context of the application that registered the event handling function

Files

Files

Name	Description
drv_adc.h	ADC Driver interface definition.
drv_adc_config_template.h	ADC Device Driver configuration template.

Description

This section lists the source and header files used by the ADC Driver Library.

drv_adc.h

ADC Driver interface definition.

Enumerations

	Name	Description
	DRV_ADC_BUFFER_STATUS	Specifies the status of the buffer for the read, write and erase operations.
	DRV_ADC_CLIENT_STATUS	Defines the client-specific status of the ADC driver.
	DRV_ADC_EVENT	Identifies the possible events that can result from a request.
	DRV_ADC_INIT_FLAGS	Identifies the initialization flags of the ADC module.

Functions

	Name	Description
	DRV_ADC_ChannelExtendedScanInputsAdd	Adds extended scan input on the supported devices. **Implementation:** Static
	DRV_ADC_ChannelExtendedScanInputsRemove	Removes extended scan input on the supported devices. **Implementation:** Static
	DRV_ADC_ChannelScanInputsAdd	Adds scan input. **Implementation:** Static
	DRV_ADC_ChannelScanInputsRemove	Removes scan input. **Implementation:** Static
	DRV_ADC_ClientStatus	Gets the current client-specific status the ADC driver. **Implementation:** Dynamic
	DRV_ADC_Close	Closes an opened-instance of the ADC driver. **Implementation:** Static/Dynamic
	DRV_ADC_Deinitialize	Deinitializes the specified instance of the ADC driver module. **Implementation:** Static/Dynamic
	DRV_ADC_Initialize	Initializes the ADC driver. **Implementation:** Static/Dynamic
	DRV_ADC_InputsRegister	Registers an input set with the driver for sampling. **Implementation:** Dynamic
	DRV_ADC_NegativeInputSelect	Selects the negative input. **Implementation:** Static
	DRV_ADC_Open	Opens the specified ADC driver instance and returns a handle to it. **Implementation:** Static/Dynamic

	DRV_ADC_PositiveInputSelect	Selects the positive input. **Implementation:** Static
	DRV_ADC_SamplesAvailable	Identifies if any the ADC driver has any samples available to read. **Implementation:** Static/Dynamic
	DRV_ADC_SamplesRead	Reads the converted sample data from the ADC driver. **Implementation:** Static/Dynamic
	DRV_ADC_SamplesReadLatest	Reads the most recently converted sample data from the ADC driver. **Implementation:** Dynamic
	DRV_ADC_Start	Starts the ADC driver sampling and converting analog to digital values. **Implementation:** Static/Dynamic
	DRV_ADC_Status	Provides the current status of the ADC driver module. **Implementation:** Dynamic
	DRV_ADC_Stop	Stops the ADC driver from sampling and converting analog to digital values. **Implementation:** Static/Dynamic
	DRV_ADC_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_ADC_INDEX_0	ADC driver index definitions.
	DRV_ADC_INDEX_1	This is macro DRV_ADC_INDEX_1.
	DRV_ADC_INDEX_2	This is macro DRV_ADC_INDEX_2.

Structures

	Name	Description
	DRV_ADC_INIT	Defines the data required to initialize or reinitialize the ADC driver.

Types

	Name	Description
	DRV_ADC_BUFFER_HANDLE	This type defines the ADC Driver Buffer handle.
	DRV_ADC_EVENT_HANDLER	Pointer to a ADC Driver Event handler function

Description

ADC Driver Interface Definition

The ADC device driver provides a simple interface to manage the ADC modules on Microchip microcontrollers. This file defines the interface definition for the ADC driver.

File Name

drv_adc.h

Company

Microchip Technology Inc.

drv_adc_config_template.h

ADC Device Driver configuration template.

Macros

	Name	Description
	_DRV_ADC_CONFIG_TEMPLATE_H	This is macro _DRV_ADC_CONFIG_TEMPLATE_H.
	DRV_ADC_ACQUISITION_TIME	Defines the acquisition time.
	DRV_ADC_ALTERNATE_INPUT_SAMPLING_ENABLE	Enable the alternate input sampling feature of the ADC.
	DRV_ADC_ANALOG_INPUT	Defines the analog input channel.
	DRV_ADC_AUTO_SAMPLING_ENABLE	Enables the auto-sampling feature of the ADC.
	DRV_ADC_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ADC_CONVERSION_CLOCK_PRESCALER	Defines the conversion clock.
	DRV_ADC_CONVERSION_CLOCK_SOURCE	Defines the conversion clock source.
	DRV_ADC_CONVERSION_TRIGGER_SOURCE	Defines the conversion trigger source.
	DRV_ADC_INDEX	ADC static index selection.
	DRV_ADC_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_ADC_INTERNAL_BUFFER_SIZE	Define the internal buffer size.
	DRV_ADC_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_ADC_INTERRUPT_SOURCE	Defines the interrupt source of the static driver.
	DRV_ADC_PERIPHERAL_ID	ADC PLIB ID Selection
	DRV_ADC_POWER_STATE	Controls the power state of the ADC.
	DRV_ADC_RESULT_FORMAT	Defines the data output format.
	DRV_ADC_SAMPLES_PER_INTERRUPT	Define the sample per interrupt.
	DRV_ADC_STOP_ON_CONVERSION_ENABLE	Enable the stop on conversion feature of the ADC.
	DRV_ADC_VOLTAGE_REFERENCE	Defines the voltage reference.

Description

ADC Device Driver Configuration Template

This header file contains the build-time configuration selections for the ADC device driver. This is the template file which give all possible configurations that can be made. This file should not be included in any project.

File Name

drv_adc_config_template.h

Company

Microchip Technology Inc.

Camera Driver Libraries

This topic describes the Camera Driver Libraries.

Introduction

This section provides information on the Camera Driver libraries that are provided in MPLAB Harmony and describes the APIs that are common to all drivers.

Library Interface

a) Common Driver Functions

	Name	Description
	DRV_CAMERA_Close	Closes an opened instance of an CAMERA module driver.
	DRV_CAMERA_Deinitialize	Deinitializes the index instance of the CAMERA module.
	DRV_CAMERA_Initialize	Initializes hardware and data for the index instance of the CAMERA module.
	DRV_CAMERA_Open	Opens the specified instance of the Camera driver for use and provides an "open-instance" handle.
	DRV_CAMERA_Reinitialize	
	DRV_CAMERA_Status	Provides the current status of the index instance of the CAMERA module.
	DRV_CAMERA_Tasks	

b) Common Data Types and Constants

	Name	Description
	DRV_CAMERA_INIT	Defines the data required to initialize or reinitialize the CAMERA driver
	DRV_CAMERA_INTERRUPT_PORT_REMAP	Defines the data required to initialize the CAMERA driver interrupt port remap.
	DRV_CAMERA_INDEX_0	Camera driver index definitions.
	DRV_CAMERA_INDEX_1	This is macro DRV_CAMERA_INDEX_1.
	DRV_CAMERA_INDEX_COUNT	Number of valid CAMERA driver indices.
	CAMERA_MODULE_ID	This is type CAMERA_MODULE_ID.

Description

Camera Driver APIs that are common to all Camera drivers.

a) Common Driver Functions

DRV_CAMERA_Close Function

Closes an opened instance of an CAMERA module driver.

File

drv_camera.h

C

```
void DRV_CAMERA_Close(DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened instance of an CAMERA module driver, making the specified handle invalid.

Preconditions

The DRV_CAMERA_Initialize routine must have been called for the specified CAMERA device instance and the DRV_CAMERA_Status must have returned SYS_STATUS_READY.

DRV_CAMERA_Open must have been called to obtain a valid opened device handle.

Example

```
myCameraHandle = DRV_CAMERA_Open(DRV_CAMERA_ID_1, DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);

DRV_CAMERA_Close(myCameraHandle);
```

Parameters

Parameters	Description
drvHandle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_CAMERA_Close (const DRV_HANDLE drvHandle)

DRV_CAMERA_Deinitialize Function

Deinitializes the index instance of the CAMERA module.

File

drv_camera.h

C

```
void DRV_CAMERA_Deinitialize(const SYS_MODULE_INDEX index);
```

Returns

None.

Description

This function deinitializes the index instance of the CAMERA module, disabling its operation (and any hardware for driver modules). It deinitializes only the specified module instance. It also resets all the internal data structures and fields for the specified instance to the default settings.

Preconditions

The DRV_CAMERA_Initialize function should have been called before calling this function.

Example

```
SYS_STATUS   cameraStatus;

DRV_CAMERA_Deinitialize(DRV_CAMERA_ID_1);

cameraStatus = DRV_CAMERA_Status(DRV_CAMERA_ID_1);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the CAMERA module to be deinitialized

Function

void DRV_CAMERA_Deinitialize (const SYS_MODULE_ID index)

DRV_CAMERA_Initialize Function

Initializes hardware and data for the index instance of the CAMERA module.

File

drv_camera.h

C

```
SYS_MODULE_OBJ DRV_CAMERA_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT *
const init);
```

Returns

None

Description

This function initializes hardware for the index instance of the CAMERA module, using the hardware initialization given data. It also initializes any internal driver data structures making the driver ready to be opened.

Preconditions

None.

Example

```
DRV_CAMERA_INIT_DATA        cameraInitData;
SYS_STATUS                  cameraStatus;

// Populate the cameraInitData structure
cameraInitData.moduleInit.powerState = SYS_MODULE_POWER_RUN_FULL;
cameraInitData.moduleInit.moduleCode = (DRV_CAMERA_INIT_DATA_MASTER | DRV_CAMERA_INIT_DATA_SLAVE);

DRV_CAMERA_Initialize(DRV_CAMERA_ID_1, (SYS_MODULE_INIT*)&cameraInitData);
cameraStatus = DRV_CAMERA_Status(DRV_CAMERA_ID_1);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the CAMERA module to be initialized
data	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and the default initialization is to be used.

Function

void DRV_CAMERA_Initialize (const CAMERA_MODULE_ID index,
const SYS_MODULE_INIT *const data)

DRV_CAMERA_Open Function

Opens the specified instance of the Camera driver for use and provides an "open-instance" handle.

File

drv_camera.h

C

```
DRV_HANDLE DRV_CAMERA_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a value identifying both the caller and the module instance). If an error occurs, the returned value is DRV_HANDLE_INVALID.

Description

This function opens the specified instance of the Camera module for use and provides a handle that is required to use the remaining driver routines.

This function opens a specified instance of the Camera module driver for use by any client module and provides an "open-instance" handle that must be provided to any of the other Camera driver operations to identify the caller and the instance of the Camera driver/hardware module.

Preconditions

The DRV_CAMERA_Initialize routine must have been called for the specified CAMERA device instance and the DRV_CAMERA_Status must have returned SYS_STATUS_READY.

Example

```
DRV_HANDLE                cameraHandle;
DRV_CAMERA_CLIENT_STATUS   cameraClientStatus;

cameraHandle = DRV_CAMERA_Open(DRV_CAMERA_ID_1, DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);
if (DRV_HANDLE_INVALID == cameraHandle)
{
    // Handle open error
}

cameraClientStatus = DRV_CAMERA_ClientStatus(cameraHandle);

// Close the device when it is no longer needed.
DRV_CAMERA_Close(cameraHandle);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the CAMERA module to be opened.
intent	Flags parameter identifying the intended usage and behavior of the driver. Multiple flags may be ORed together to specify the intended usage of the device. See the DRV_IO_INTENT definition.

Function

DRV_HANDLE DRV_CAMERA_Open (const SYS_MODULE_INDEX index,
const DRV_IO_INTENT intent)

DRV_CAMERA_Reinitialize Function

File

drv_camera.h

C

```
void DRV_CAMERA_Reinitialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT *const data);
```

Returns

None.

Preconditions

The DRV_CAMERA_Initialize function should have been called before calling this function.

Example

```
SYS_MODULE_INIT cameraInit;
SYS_STATUS      cameraStatus;

DRV_CAMERA_Reinitialize(DRV_CAMERA_ID_1, &cameraStatus);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the CAMERA module to be reinitialized
data	Pointer to the data structure containing any data necessary to reinitialize the hardware. This pointer may be null if no data is required and default configuration is to be used.

Function

void DRV_CAMERA_Reinitialize(const SYS_MODULE_ID index,

const SYS_MODULE_INIT *const data)

DRV_CAMERA_Status Function

Provides the current status of the index instance of the CAMERA module.

File

drv_camera.h

C

```
SYS_STATUS DRV_CAMERA_Status(const SYS_MODULE_INDEX index);
```

Description

This function provides the current status of the index instance of the CAMERA module.

Preconditions

The DRV_CAMERA_Initialize function should have been called before calling this function.

Function

SYS_STATUS DRV_CAMERA_Status (const CAMERA_MODULE_ID index)

DRV_CAMERA_Tasks Function

File

drv_camera.h

C

```
void DRV_CAMERA_Tasks(SYS_MODULE_OBJ object);
```

Section

Camera Driver Client Status Functions

b) Common Data Types and Constants

DRV_CAMERA_INIT Structure

Defines the data required to initialize or reinitialize the CAMERA driver

File

drv_camera.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  int cameraId;
  SYS_MODULE_OBJ (* drvInitialize)(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const
init);
  DRV_HANDLE (* drvOpen)(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
  INT_SOURCE interruptSource;
  DRV_CAMERA_INTERRUPT_PORT_REMAP interruptPort;
  uint16_t orientation;
  uint16_t horizontalResolution;
  uint16_t verticalResolution;
} DRV_CAMERA_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
int cameraId;	ID
uint16_t orientation;	Orientation of the display (given in degrees of 0,90,180,270)
uint16_t horizontalResolution;	Horizontal Resolution of the displayed orientation in Pixels

Description

CAMERA Driver Initialization Data

This data type defines the data required to initialize or reinitialize the CAMERA driver. If the driver is built statically, the members of this data structure are statically over-ridden by static override definitions in the system_config.h file.

Remarks

None.

DRV_CAMERA_INTERRUPT_PORT_REMAP Structure

Defines the data required to initialize the CAMERA driver interrupt port remap.

File

drv_camera.h

C

```
typedef struct {
  PORTS_REMAP_INPUT_FUNCTION inputFunction;
  PORTS_REMAP_INPUT_PIN inputPin;
  PORTS_ANALOG_PIN analogPin;
  PORTS_PIN_MODE pinMode;
  PORTS_CHANNEL channel;
  PORTS_DATA_MASK dataMask;
} DRV_CAMERA_INTERRUPT_PORT_REMAP;
```

Description

CAMERA Driver Interrupt Port Remap Initialization Data

This data type defines the data required to initialize the CAMERA driver interrupt port remap.

Remarks

None.

DRV_CAMERA_INDEX_0 Macro

Camera driver index definitions.

File

drv_camera.h

C

```
#define DRV_CAMERA_INDEX_0 0
```

Description

Camera Driver Module Index Numbers

These constants provide the Camera driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_CAMERA_Initialize and DRV_CAMERA_Open functions to identify the driver instance in use.

DRV_CAMERA_INDEX_1 Macro

File

drv_camera.h

C

```
#define DRV_CAMERA_INDEX_1 1
```

Description

This is macro DRV_CAMERA_INDEX_1.

DRV_CAMERA_INDEX_COUNT Macro

Number of valid CAMERA driver indices.

File

drv_camera.h

C

```
#define DRV_CAMERA_INDEX_COUNT 1
```

Description

CAMERA Driver Module Index Count

This constant identifies the number of valid CAMERA driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific header files defined as part of the peripheral libraries.

CAMERA_MODULE_ID Enumeration

File

drv_camera.h

C

```
typedef enum {
  CAMERA_MODULE_OVM7690
} CAMERA_MODULE_ID;
```

Description

This is type CAMERA_MODULE_ID.

Files

Files

Name	Description
drv_camera.h	Camera device driver interface file.

Description

drv_camera.h

Camera device driver interface file.

Enumerations

	Name	Description
	CAMERA_MODULE_ID	This is type CAMERA_MODULE_ID.

Functions

	Name	Description
	DRV_CAMERA_Close	Closes an opened instance of an CAMERA module driver.
	DRV_CAMERA_Deinitialize	Deinitializes the index instance of the CAMERA module.
	DRV_CAMERA_Initialize	Initializes hardware and data for the index instance of the CAMERA module.
	DRV_CAMERA_Open	Opens the specified instance of the Camera driver for use and provides an "open-instance" handle.
	DRV_CAMERA_Reinitialize	
	DRV_CAMERA_Status	Provides the current status of the index instance of the CAMERA module.
	DRV_CAMERA_Tasks	

Macros

	Name	Description
	DRV_CAMERA_INDEX_0	Camera driver index definitions.
	DRV_CAMERA_INDEX_1	This is macro DRV_CAMERA_INDEX_1.
	DRV_CAMERA_INDEX_COUNT	Number of valid CAMERA driver indices.

Structures

	Name	Description
	DRV_CAMERA_INIT	Defines the data required to initialize or reinitialize the CAMERA driver
	DRV_CAMERA_INTERRUPT_PORT_REMAP	Defines the data required to initialize the CAMERA driver interrupt port remap.

Description

Camera Driver Interface

The Camera driver provides a abstraction to all camera drivers.

File Name

drv_camera.h

Company

Microchip Technology Inc.

OVM7690 Camera Driver Library

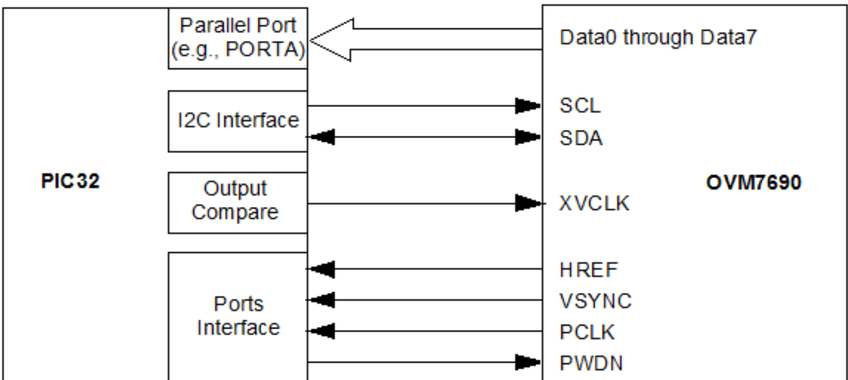
This topic describes the OVM7690 Camera Driver Library.

Introduction

The OVM7690 Camera Driver provides a high-level interface to manage the OmniVision Technologies, Inc. OVM7690 640x480 CameraCube™ device (referred to as the OVM7690) that is interfaced with serial and parallel ports to a Microchip microcontroller for providing camera solutions.

Description

The OVM7690 640x480 CameraCube™ device (referred to as the OVM7690) can be interfaced to a Microchip microcontroller using the I2C serial interface and parallel port interface. The I2C serial interface is used for control command transfer. The I2C module from the microcontroller is connected to the SCCB serial interface of the OVM7690. The parallel port interface is used to transfer pixel data from the OVM7690 to the microcontroller. There are few other signals from the camera to be interfaced with the microcontroller. The XVCLK pin of the camera is driven by the Output Compare module. Frame synchronization signals such as HREF and VSYNC from the camera are connected to suitable pins supporting change notification within the microcontroller. The PCLK pin of the camera drives the pixel clock and is connected at the pin of the microcontroller supporting external interrupts. The PWDN pin of the camera supports camera power-down mode and is connected at any output port pin of the microcontroller. A typical interface of the OVM7690 to a PIC32 device is provided in the following diagram:



Using the Library

This topic describes the basic architecture of the OVM7690 Camera Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_camera_ovm7690.h`

The interface to the Camera Driver Library is defined in the `drv_camera_ovm7690.h` header file.

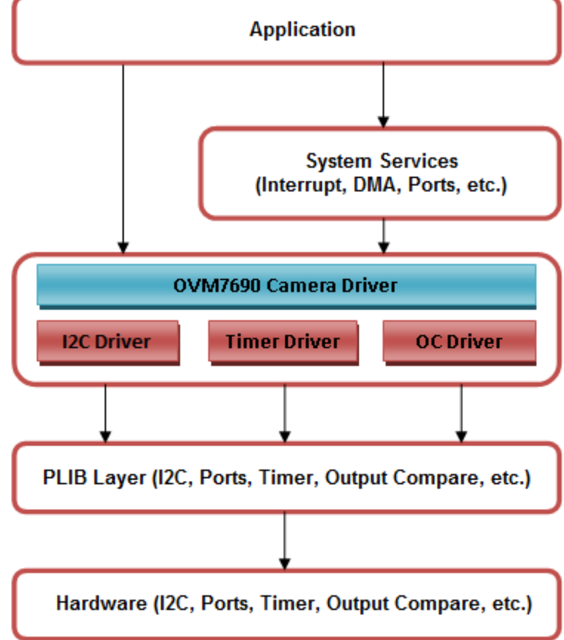
Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the OVM7690 Camera Driver Library on Microchip's microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The OVM7690 Camera Driver is modeled using the abstraction model, as shown in the following diagram.



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address the overall operation of the OVM7690 Camera Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization and deinitialization.
Client Setup Functions	Provides open and close functions.
Camera-specific Functions	Provides APIs that are camera-specific.

© 2013-2015 Microchip Technology Inc.

Other Functions	Provides miscellaneous driver-specific functions such as register set functions, among others.

How the Library Works

Provides information on how the OVM7690 Camera Driver Library works.

Description

The library provides interfaces to support:

- System functionality
- Client functionality

System Initialization

The system performs the Initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the OVM7690 would be initialized with the following configuration settings that are supported by the specific OVM7690 device hardware:

- Camera ID: OVM7690 ID
- Source Port: Address of source port to which the pixel data is received
- Horizontal Sync Channel: Channel of the pin to be configured as horizontal sync
- Horizontal Sync Position: Horizontal sync port pin position from selected port channel
- Vertical Sync Channel: Channel the pin to be configured as vertical sync
- Vertical Sync Position: Vertical sync port pin position from selected port channel
- Horizontal Sync Interrupt Source
- Vertical Sync Interrupt Source
- DMA Channel: DMA channel to transfer pixel data from camera to frame buffer
- DMA Channel Trigger Source
- Bits Per Pixel: Bits per pixel to define the size of frame line

The DRV_CAMERA_OVM7690_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handler returned by the Initialize Interface would be used by the other interfaces such as DRV_CAMERA_OVM7690_Deinitialize.

Client Access

For the application to start using an instance of the module, it must call the DRV_CAMERA_OVM7690_Open function. The DRV_CAMERA_OVM7690_Open function provides a driver handle to the OVM7690 Camera Driver instance for operations. If the driver is deinitialized using the function DRV_CAMERA_OVM7690_Deinitialize function, the application must call the DRV_CAMERA_OVM7690_Open function again to set up the instance of the driver.

Client Operations

Client operations provide the API interface for control command and pixel data transfer from camera OVM7690 to the Graphics Frame Buffer.

Configuring the Library

Macros

	Name	Description
	DRV_OVM7690_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.

Description

The configuration of the OVM7690 Camera Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the OVM7690 Camera Driver build. Based on the selections made here and the system setup, the OVM7690 Camera Driver may support the selected features. These configuration settings will apply to all instances of the driver.

This header can be placed anywhere in the application specific folders and the path of this header needs to be presented to the include search for a successful build. Refer to the Applications Overview section for more details.

Control Commands

The following Camera OVM7690 specific control commands are provided:

- DRV_CAMERA_OVM7690_FrameBufferAddressSet
- DRV_CAMERA_OVM7690_Start
- DRV_CAMERA_OVM7690_Stop
- DRV_CAMERA_OVM7690_FrameRectSet

Application Process

An application needs to perform following steps:

1. The system should have completed necessary setup initializations.
2. The I2C driver object should have been initialized by calling DRV_I2C_Initialize.
3. The Timer driver object should have been initialized by calling DRV_Timer_Initialize,
4. The Output Control driver object should have been initialized by calling DRV_OC_Initialize,
5. The Camera OVM7690 driver object should have been initialized by calling DRV_CAMERA_OVM7690_Initialize,
6. Open the Camera OVM7690 driver client by calling DRV_CAMERA_OVM7690_Open.
7. Pass the Graphics Frame buffer address to Camera OVM7690 Driver by calling DRV_CAMERA_OVM7690_FrameBufferAddressSet.
8. Set the Frame Rectangle area by calling DRV_CAMERA_OVM7690_FrameRectSet.
9. Set Other Camera settings such as: soft reset, enabling pclk, enabling href, enabling vsync, output color format, reversing HREF polarity, gating clock to the HREF, pixel clock frequency, sub-sampling mode by calling DRV_CAMERA_OVM7690_RegisterSet.
10. Start the Camera OVM7690 by calling DRV_CAMERA_OVM7690_Start.

DRV_OVM7690_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_ovm7690_config_template.h

C

```
#define DRV_OVM7690_INTERRUPT_MODE false
```

Description

OVM7690 Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of OVM7690 operation is desired
- false - Select if polling mode of OVM7690 operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

Building the Library

This section lists the files that are available in the OVM7690 Camera Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/camera/ovm7690`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_camera_ovm7690.h	This file provides the interface definitions of the OVM7690 Camera Driver.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/drv_camera_ovm7690.c	This file contains the implementation of the OVM7690 Camera Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The OVM7690 Camera Driver Library depends on the following modules:

- I2C Driver Library
- Output Compare Driver Library
- Timer Driver Library

Library Interface

a) System Functions

	Name	Description
	DRV_CAMERA_OVM7690_Initialize	Initializes the Camera OVM7690 instance for the specified driver index.
	DRV_CAMERA_OVM7690_Deinitialize	Deinitializes the specified instance of the Camera OVM7690 driver module.
	DRV_CAMERA_OVM7690_RegisterSet	Sets the camera OVM7690 configuration registers

b) Client Setup Functions

	Name	Description
	DRV_CAMERA_OVM7690_Open	Opens the specified Camera OVM7690 driver instance and returns a handle to it.
	DRV_CAMERA_OVM7690_Close	Closes an opened-instance of the Camera OVM7690 driver.

c) Camera-specific Functions

	Name	Description
	DRV_CAMERA_OVM7690_FrameBufferAddressSet	Sets Frame buffer address.
	DRV_CAMERA_OVM7690_Start	Starts camera rendering to the display.
	DRV_CAMERA_OVM7690_Stop	Stops rendering the camera Pixel data.
	DRV_CAMERA_OVM7690_FrameRectSet	Sets the Frame Rectangle Set.

d) Other Functions

	Name	Description
	DRV_CAMERA_OVM7690_HsyncEventHandler	Horizontal Synchronization Event Handler
	DRV_CAMERA_OVM7690_VsyncEventHandler	Vertical Synchronization Event Handler

e) Data Types and Constants

	Name	Description
	DRV_CAMERA_OVM7690_CLIENT_OBJ	Camera OVM7690 Driver Client Object.
	DRV_CAMERA_OVM7690_CLIENT_STATUS	Identifies Camera OVM7690 possible client status.
	DRV_CAMERA_OVM7690_ERROR	Identifies Camera OVM7690 possible errors.
	DRV_CAMERA_OVM7690_INIT	Camera OVM7690 Driver Initialization Parameters.
	DRV_CAMERA_OVM7690_OBJ	Camera OVM7690 Driver Instance Object.
	DRV_CAMERA_OVM7690_RECT	Camera OVM7690 Window Rectangle co-ordinates.
	DRV_CAMERA_OVM7690_REG12_OP_FORMAT	List of Camera OVM7690 Device Register Addresses.
	DRV_CAMERA_OVM7690_INDEX_0	OVM7690 driver index definitions
	DRV_CAMERA_OVM7690_INDEX_1	This is macro DRV_CAMERA_OVM7690_INDEX_1.

	DRV_CAMERA_OVM7690_REG12_SOFT_RESET	Driver Camera OVM7690 Register 0x12 Soft reset flag.
	DRV_CAMERA_OVM7690_SCCB_READ_ID	Camera OVM7690 SCCB Interface device Read Slave ID.
	DRV_CAMERA_OVM7690_SCCB_WRITE_ID	Camera OVM7690 SCCB Interface device Write Slave ID

Description

This section describes the Application Programming Interface (API) functions of the Camera Driver Library.

a) System Functions

DRV_CAMERA_OVM7690_Initialize Function

Initializes the Camera OVM7690 instance for the specified driver index.

File

drv_camera_ovm7690.h

C

```
SYS_MODULE_OBJ DRV_CAMERA_OVM7690_Initialize(const SYS_MODULE_INDEX drvIndex, const
SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the Camera OVM7690 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the Camera OVM7690 module ID. Refer to the description of the DRV_CAMERA_OVM7690_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other OVM7690 routine is called.

This routine should only be called once during system initialization unless DRV_CAMERA_OVM7690_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
// The following code snippet shows an example OVM7690 driver initialization.

DRV_CAMERA_OVM7690_INIT      cameraInit;
SYS_MODULE_OBJ               objectHandle;

cameraInit.cameraID                    = CAMERA_MODULE_OVM7690;
cameraInit.sourcePort                  = (void *)&PORTK,
cameraInit.hsyncInterruptSource        = INT_SOURCE_CHANGE_NOTICE_A,
cameraInit.vsyncInterruptSource        = INT_SOURCE_CHANGE_NOTICE_J,
cameraInit.dmaChannel                  = DRV_CAMERA_OVM7690_DMA_CHANNEL_INDEX,
cameraInit.dmaTriggerSource            = DMA_TRIGGER_EXTERNAL_2,
cameraInit.bpp                         = GFX_CONFIG_COLOR_DEPTH,

objectHandle = DRV_CAMERA_OVM7690_Initialize( DRV_CAMERA_OVM7690_INDEX_0,
                                              (SYS_MODULE_INIT*)&cameraInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized

init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_CAMERA_OVM7690_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

)

DRV_CAMERA_OVM7690_Deinitialize Function

Deinitializes the specified instance of the Camera OVM7690 driver module.

File

drv_camera_ovm7690.h

C

```
void DRV_CAMERA_OVM7690_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the Camera OVM7690 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_CAMERA_OVM7690_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ       object; //  Returned from DRV_CAMERA_OVM7690_Initialize
SYS_STATUS           status;

DRV_CAMERA_OVM7690_Deinitialize(object);

status = DRV_CAMERA_OVM7690_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_CAMERA_OVM7690_Initialize routine

Function

void DRV_CAMERA_OVM7690_Deinitialize(SYS_MODULE_OBJ object)

DRV_CAMERA_OVM7690_RegisterSet Function

Sets the camera OVM7690 configuration registers

File

drv_camera_ovm7690.h

C

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_RegisterSet(DRV_CAMERA_OVM7690_REGISTER_ADDRESS
regIndex, uint8_t regValue);
```

Returns

DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE - Invalid driver Handle.
DRV_CAMERA_OVM7690_ERROR_NONE - No error.

Description

This routine sets the Camera OVM7690 configuration registers using SCCB interface.

Remarks

This routine can be used separately or within a interface.

Preconditions

The DRV_CAMERA_OVM7690_Initialize routine must have been called for the specified Camera OVM7690 driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

The SCCB interface also must have been initialized to configure the Camera OVM7690.

Example

```
DRV_HANDLE handle;
uint8_t reg12 = DRV_CAMERA_OVM7690_REG12_SOFT_RESET;

 handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
 if (DRV_HANDLE_INVALID == handle)
 {
     //error
     return;
 }

 if ( DRV_CAMERA_OVM7690_RegisterSet( DRV_CAMERA_OVM7690_REG12_REG_ADDR,
                                      reg12 ) !=
                                      DRV_CAMERA_OVM7690_ERROR_NONE )
 {
     //error
     return;
 }
```

Parameters

Parameters	Description
regIndex	Defines the configuration register addresses for OVM7690.
regValue	Defines the register value to be set.

Function

DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_RegisterSet

(

DRV_CAMERA_OVM7690_REGISTER_ADDRESS regIndex,

```
uint8_t regValue
)
```

b) Client Setup Functions

DRV_CAMERA_OVM7690_Open Function

Opens the specified Camera OVM7690 driver instance and returns a handle to it.

File

drv_camera_ovm7690.h

C

```
DRV_HANDLE DRV_CAMERA_OVM7690_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT
ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_CAMERA_OVM7690_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver is not ready to be opened, typically when the initialize routine has not completed execution.

Description

This routine opens the specified Camera OVM7690 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

Remarks

The handle returned is valid until the DRV_CAMERA_OVM7690_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application.

Preconditions

Function DRV_CAMERA_OVM7690_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_CAMERA_OVM7690_Open
(
const SYS_MODULE_INDEX index,
const DRV_IO_INTENT ioIntent
)

DRV_CAMERA_OVM7690_Close Function

Closes an opened-instance of the Camera OVM7690 driver.

File

drv_camera_ovm7690.h

C

```
void DRV_CAMERA_OVM7690_Close(DRV_HANDLE handle);
```

Returns

None.

Description

This routine closes an opened-instance of the Camera OVM7690 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines (with one possible exception described in the "Remarks" section). A new handle must be obtained by calling DRV_CAMERA_OVM7690_Open before the caller may use the driver again

Remarks

Usually there is no need for the client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_CAMERA_OVM7690_Initialize routine must have been called for the specified Camera OVM7690 driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_USART_Open

DRV_CAMERA_OVM7690_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_CAMERA_OVM7690_Close(DRV_Handle handle)

c) Camera-specific Functions

DRV_CAMERA_OVM7690_FrameBufferAddressSet Function

Sets Frame buffer address.

File

drv_camera_ovm7690.h

C

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_FrameBufferAddressSet(DRV_HANDLE handle, void *
frameBuffer);
```

Returns

DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE - Invalid driver Handle.
DRV_CAMERA_OVM7690_ERROR_NONE - No error.

Description

This routine will set the Frame Buffer Address. This frame buffer address will point to the location at which frame data is to be rendered. This buffer is shared with the display controller to display the frame on the display.

Remarks

This routine is mandatory. A valid frame buffer address need to be set to display camera data.

Preconditions

The DRV_CAMERA_OVM7690_Initialize routine must have been called for the specified Camera OVM7690 driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;
uint16_t frameBuffer[DISP_VER_RESOLUTION][DISP_HOR_RESOLUTION];

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
    return;
}

if ( DRV_CAMERA_OVM7690_FrameBufferAddressSet( handle, (void *) frameBuffer ) !=
                                    DRV_CAMERA_OVM7690_ERROR_NONE )
{
    //error
    return;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_FrameBufferAddressSet

(

DRV_HANDLE handle,

void * frameBuffer
)

DRV_CAMERA_OVM7690_Start Function

Starts camera rendering to the display.

File

drv_camera_ovm7690.h

C

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_Start(DRV_HANDLE handle);
```

Returns

DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE - Invalid driver Handle.
DRV_CAMERA_OVM7690_ERROR_NONE - No error.

Description

This routine starts the camera rendering to the display by writing the pixel data to the frame buffer. Frame buffer is shared between camera OVM7690 and display controller.

Remarks

This routine is mandatory. Camera module will not update the framebuffer without calling this routine.

Preconditions

The DRV_CAMERA_OVM7690_Initialize routine must have been called for the specified Camera OVM7690 driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

DRV_CAMERA_OVM7690_FrameBufferAddressSet must have been called to set a valid frame buffer address.

Example

```
DRV_HANDLE handle;
uint16_t frameBuffer[DISP_VER_RESOLUTION][DISP_HOR_RESOLUTION];

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
    return;
}

if ( DRV_CAMERA_OVM7690_FrameBufferAddressSet( handle, (void *) frameBuffer ) !=
                                    DRV_CAMERA_OVM7690_ERROR_NONE )
{
    //error
    return;
}

if ( DRV_CAMERA_OVM7690_Start( handle ) !=
                                    DRV_CAMERA_OVM7690_ERROR_NONE )
{
    //error
    return;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_Start
(
DRV_HANDLE handle
);

DRV_CAMERA_OVM7690_Stop Function

Stops rendering the camera Pixel data.

File

drv_camera_ovm7690.h

C

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_Stop(DRV_HANDLE handle);
```

Returns

DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE - Invalid driver Handle.
DRV_CAMERA_OVM7690_ERROR_NONE - No error.

Description

This routine starts the camera rendering to the display by writing the pixel data to the frame buffer. Frame buffer is shared between camera OVM7690 and display controller.

Remarks

This routine only disables the interrupt for hsync and vsync. To stop the camera the power down pin need to be toggled to active high value. This will stop the camera internal clock and will maintain the register values.

Preconditions

The DRV_CAMERA_OVM7690_Initialize routine must have been called for the specified Camera OVM7690 driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
    return;
}

if ( DRV_CAMERA_OVM7690_Stop( handle ) !=
                                    DRV_CAMERA_OVM7690_ERROR_NONE )
{
    //error
    return;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine.

Function

DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_Stop
(
DRV_HANDLE handle
);

DRV_CAMERA_OVM7690_FrameRectSet Function

Sets the Frame Rectangle Set.

File

drv_camera_ovm7690.h

C

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_FrameRectSet(DRV_HANDLE handle, uint32_t left,
uint32_t top, uint32_t right, uint32_t bottom);
```

Returns

DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE - Invalid driver Handle.
DRV_CAMERA_OVM7690_ERROR_NONE - No error.

Description

This routine sets the frame rectangle coordinates. The Frame within the rectangle is copied to the frame buffer. The left and top values are expected to be less than right and bottom respectively. Left, top, right and bottom values are also expected to be within range of screen coordinates. Internally it calls DRV_CAMERA_OVM7690_RegisterSet routine to set respective registers. The rectangle coordinates are also maintained in the driver object.

Remarks

This functional is option if default values are expected to be used.

Preconditions

The DRV_CAMERA_OVM7690_Initialize routine must have been called for the specified Camera OVM7690 driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

The SCCB interface also must have been initialized to configure the Camera OVM7690.

Example

```
DRV_HANDLE handle;
uint32_t left   = 0x69;
uint32_t top    = 0x0E;
uint32_t right  = DISP_HOR_RESOLUTION + 0x69;
uint32_t bottom = DISP_VER_RESOLUTION + 0x69;

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
    return;
}

if ( DRV_CAMERA_OVM7690_FrameRectSet( handle, left, top, right, bottom ) !=
                                     DRV_CAMERA_OVM7690_ERROR_NONE )
{
    //error
    return;
```

}

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
left	left frame coordinate
top	top frame coordinate
right	right frame coordinate
bottom	bottom frame coordinate

Function

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_FrameRectSet
(
    DRV_HANDLE handle,
uint32_t   left,
uint32_t   top,
uint32_t   right,
uint32_t   bottom
)
```

d) Other Functions

DRV_CAMERA_OVM7690_HsyncEventHandler Function

Horizontal Synchronization Event Handler

File

drv_camera_ovm7690.h

C

```
void DRV_CAMERA_OVM7690_HsyncEventHandler(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is called when Camera OVM7690 sends Horizontal Sync Pulse on HSync line. It sets the next line address in the DMA module.

Remarks

This routine is mandatory.

Preconditions

The DRV_CAMERA_OVM7690_Initialize routine must have been called for the specified Camera OVM7690 driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_CAMERA_OVM7690_INIT      cameraInit;
SYS_MODULE_OBJ               objectHandle;

cameraInit.cameraID                  = CAMERA_MODULE_OVM7690;
cameraInit.sourcePort                = (void *)&PORTK,
cameraInit.hsyncInterruptSource      = INT_SOURCE_CHANGE_NOTICE_A,
cameraInit.vsyncInterruptSource      = INT_SOURCE_CHANGE_NOTICE_J,
cameraInit.dmaChannel                = DRV_CAMERA_OVM7690_DMA_CHANNEL_INDEX,
cameraInit.dmaTriggerSource          = DMA_TRIGGER_EXTERNAL_2,
cameraInit.bpp                       = GFX_CONFIG_COLOR_DEPTH,

objectHandle = DRV_CAMERA_OVM7690_Initialize( DRV_CAMERA_OVM7690_INDEX_0,
                                              (SYS_MODULE_INIT*)&cameraInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
    return;
}

void __ISR( HSYNC_ISR_VECTOR) _Ovm7690HSyncHandler(void)
{
    DRV_CAMERA_OVM7690_HsyncEventHandler(objectHandle);

    SYS_INT_SourceStatusClear(HSYNC_INTERRUPT_SOURCE);
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_CAMERA_VOM7690_Initialize routine

Function

void DRV_CAMERA_OVM7690_HsyncEventHandler(SYS_MODULE_OBJ object)

DRV_CAMERA_OVM7690_VsyncEventHandler Function

Vertical Synchronization Event Handler

File

drv_camera_ovm7690.h

C

```
void DRV_CAMERA_OVM7690_VsyncEventHandler(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is called when Camera OVM7690 sends Vertical Sync Pulse on VSync line. It clears the number of lines drawn variable.

Remarks

This routine is mandatory.

Preconditions

The DRV_CAMERA_OVM7690_Initialize routine must have been called for the specified Camera OVM7690 driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_CAMERA_OVM7690_INIT      cameraInit;
SYS_MODULE_OBJ               objectHandle;

cameraInit.cameraID                   = CAMERA_MODULE_OVM7690;
cameraInit.sourcePort                 = (void *)&PORTK,
cameraInit.hsyncInterruptSource       = INT_SOURCE_CHANGE_NOTICE_A,
cameraInit.vsyncInterruptSource       = INT_SOURCE_CHANGE_NOTICE_J,
cameraInit.dmaChannel                 = DRV_CAMERA_OVM7690_DMA_CHANNEL_INDEX,
cameraInit.dmaTriggerSource           = DMA_TRIGGER_EXTERNAL_2,
cameraInit.bpp                        = GFX_CONFIG_COLOR_DEPTH,

objectHandle = DRV_CAMERA_OVM7690_Initialize( DRV_CAMERA_OVM7690_INDEX_0,
                                              (SYS_MODULE_INIT*)&cameraInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
```

```
    return;
}

void __ISR( VSYNC_ISR_VECTOR) _Ovm7690VSyncHandler(void)
{
    DRV_CAMERA_OVM7690_VsyncEventHandler(objectHandle);

    SYS_INT_SourceStatusClear(VSYNC_INTERRUPT_SOURCE);
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_CAMERA_OVM7690_Initialize routine

Function

void DRV_CAMERA_OVM7690_VsyncEventHandler(SYS_MODULE_OBJ object)

e) Data Types and Constants

DRV_CAMERA_OVM7690_CLIENT_OBJ Structure

Camera OVM7690 Driver Client Object.

File

drv_camera_ovm7690.h

C

```
typedef struct {
  DRV_CAMERA_OVM7690_OBJ * hDriver;
  DRV_IO_INTENT ioIntent;
  bool inUse;
  DRV_CAMERA_OVM7690_ERROR error;
  DRV_CAMERA_OVM7690_CLIENT_STATUS status;
} DRV_CAMERA_OVM7690_CLIENT_OBJ;
```

Members

Members	Description
DRV_CAMERA_OVM7690_OBJ * hDriver;	The hardware instance object associated with the client
DRV_IO_INTENT ioIntent;	The IO intent with which the client was opened
bool inUse;	This flags indicates if the object is in use or is • available
DRV_CAMERA_OVM7690_ERROR error;	Driver Error
DRV_CAMERA_OVM7690_CLIENT_STATUS status;	client status

Description

Camera OVM7690 Driver Client Object.

This structure provides Camera OVM7690 driver client object

Remarks

These values are been updated into the DRV_CAMERA_OVM7690_Open routine.

DRV_CAMERA_OVM7690_CLIENT_STATUS Enumeration

Identifies Camera OVM7690 possible client status.

File

drv_camera_ovm7690.h

C

```
typedef enum {
  DRV_CAMERA_OVM7690_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR,
  DRV_CAMERA_OVM7690_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED,
  DRV_CAMERA_OVM7690_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  DRV_CAMERA_OVM7690_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY
} DRV_CAMERA_OVM7690_CLIENT_STATUS;
```

Members

Members	Description
DRV_CAMERA_OVM7690_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR	An error has occurred.
DRV_CAMERA_OVM7690_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED	The driver is closed, no operations for this client are ongoing, and/or the given handle is invalid.
DRV_CAMERA_OVM7690_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	The driver is currently busy and cannot start additional operations.
DRV_CAMERA_OVM7690_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY	The module is running and ready for additional operations

Description

Camera OVM7690 Client Status.

This enumeration defines possible Camera OVM7690 possible Client Status.

Remarks

This enumeration values are set by driver interfaces: DRV_CAMERA_OVM7690_Open and DRV_CAMERA_OVM7690_Close.

DRV_CAMERA_OVM7690_ERROR Enumeration

Identifies Camera OVM7690 possible errors.

File

drv_camera_ovm7690.h

C

```
typedef enum {
  DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE,
  DRV_CAMERA_OVM7690_ERROR_NONE
} DRV_CAMERA_OVM7690_ERROR;
```

Members

Members	Description
DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE	Camera OVM7690 Driver Invalid Handle
DRV_CAMERA_OVM7690_ERROR_NONE	Camear OVM7690 Driver error none

Description

Camera OVM7690 Error flag

This enumeration defines possible Camera OVM7690 possible errors.

Remarks

This enumeration values are returned by driver interfaces in case of errors.

DRV_CAMERA_OVM7690_INIT Structure

Camera OVM7690 Driver Initialization Parameters.

File

drv_camera_ovm7690.h

C

```
typedef struct {
  CAMERA_MODULE_ID cameraID;
  void * sourcePort;
  PORTS_CHANNEL hsyncChannel;
  PORTS_BIT_POS hsyncPosition;
  PORTS_CHANNEL vsyncChannel;
  PORTS_BIT_POS vsyncPosition;
  INT_SOURCE hsyncInterruptSource;
  INT_SOURCE vsyncInterruptSource;
  DMA_CHANNEL dmaChannel;
  DMA_TRIGGER_SOURCE dmaTriggerSource;
  uint16_t bpp;
} DRV_CAMERA_OVM7690_INIT;
```

Members

Members	Description
CAMERA_MODULE_ID cameraID;	camera module ID
void * sourcePort;	source Port Address
PORTS_CHANNEL hsyncChannel;	hsync pin channel
PORTS_BIT_POS hsyncPosition;	hsync pin bit position
PORTS_CHANNEL vsyncChannel;	vsync pin channel
PORTS_BIT_POS vsyncPosition;	vsync pin bit position
INT_SOURCE hsyncInterruptSource;	hsync Interrupt Source
INT_SOURCE vsyncInterruptSource;	vsync Interrupt Source
DMA_CHANNEL dmaChannel;	dma channel
DMA_TRIGGER_SOURCE dmaTriggerSource;	dma trigger source
uint16_t bpp;	bits per pixel

Description

Camera OVM7690 Initialization parameters

This structure defines Camera OVM7690 Driver Initialization Parameters.

Remarks

These values should be passed into the DRV_CAMERA_OVM7690_Initialize routine.

DRV_CAMERA_OVM7690_OBJ Structure

Camera OVM7690 Driver Instance Object.

File

drv_camera_ovm7690.h

C

```
typedef struct {
  CAMERA_MODULE_ID moduleId;
  SYS_STATUS status;
  bool inUse;
  bool isExclusive;
  size_t nClients;
  PORTS_CHANNEL hsyncChannel;
  PORTS_BIT_POS hsyncPosition;
  PORTS_CHANNEL vsyncChannel;
  PORTS_BIT_POS vsyncPosition;
```

```
    INT_SOURCE hsyncInterruptSource;
    INT_SOURCE vsyncInterruptSource;
    SYS_DMA_CHANNEL_HANDLE dmaHandle;
    DMA_CHANNEL dmaChannel;
    DMA_TRIGGER_SOURCE dmaTriggerSource;
    bool dmaTransferComplete;
    void * sourcePort;
    uint32_t frameLineCount;
    uint32_t frameLineSize;
    void * frameLineAddress;
    void * frameBufferAddress;
    DRV_CAMERA_OVM7690_RECT rect;
    uint16_t bpp;
} DRV_CAMERA_OVM7690_OBJ;
```

Members

Members	Description
CAMERA_MODULE_ID moduleId;	The module index associated with the object
SYS_STATUS status;	The status of the driver
bool inUse;	Flag to indicate this object is in use
bool isExclusive;	Flag to indicate that driver has been opened exclusively.
size_t nClients;	Keeps track of the number of clients • that have opened this driver
PORTS_CHANNEL hsyncChannel;	hsync pin channel
PORTS_BIT_POS hsyncPosition;	hsync pin bit position
PORTS_CHANNEL vsyncChannel;	vsync pin channel
PORTS_BIT_POS vsyncPosition;	vsync pin bit position
INT_SOURCE hsyncInterruptSource;	hsync Interrupt Source
INT_SOURCE vsyncInterruptSource;	vsync Interrupt Source
SYS_DMA_CHANNEL_HANDLE dmaHandle;	dma Handle
DMA_CHANNEL dmaChannel;	Read DMA channel
DMA_TRIGGER_SOURCE dmaTriggerSource;	dma Trigger Source
bool dmaTransferComplete;	dma Transfer complete flag
void * sourcePort;	source port address
uint32_t frameLineCount;	frame Line Count
uint32_t frameLineSize;	frame Line Size
void * frameLineAddress;	frame Line Address
void * frameBufferAddress;	framebuffer address
DRV_CAMERA_OVM7690_RECT rect;	Window Rect
uint16_t bpp;	Bits Per Pixel supported

Description

Camera OVM7690 Driver Instance Object

This structure provide definition of Camera OVM7690 Driver Instance Object.

Remarks

These values are been updated into the DRV_CAMERA_OVM7690_Initialize routine.

DRV_CAMERA_OVM7690_RECT Structure

Camera OVM7690 Window Rectangle co-ordinates.

File

drv_camera_ovm7690.h

C

```
typedef struct {
  uint32_t left;
  uint32_t top;
  uint32_t right;
  uint32_t bottom;
} DRV_CAMERA_OVM7690_RECT;
```

Members

Members	Description
uint32_t left;	Camera OVM7690 Window left co-ordinate
uint32_t top;	Camera OVM7690 Window top co-ordinate
uint32_t right;	Camera OVM7690 Window right co-ordinate
uint32_t bottom;	Camera OVM7690 Window bottom co-ordinate

Description

Camera OVM7690 Window Rect

This structure defines Window Rectangle co-ordinates as left, right, top and bottom.

Remarks

These values should be passed into the DRV_CAMERA_OVM7690_FrameRectSet routine.

DRV_CAMERA_OVM7690_REG12_OP_FORMAT Enumeration

List of Camera OVM7690 Device Register Addresses.

File

drv_camera_ovm7690.h

C

```
typedef enum {
  DRV_CAMERA_OVM7690_REG12_OP_FORMAT_RAW_2
} DRV_CAMERA_OVM7690_REG12_OP_FORMAT;
```

Members

Members	Description
DRV_CAMERA_OVM7690_REG12_OP_FORMAT_RAW_2	Bayer Raw Format

Description

Camera OVM7690 Device Register Addresses.

This enumeration defines list of device register addresses.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_CAMERA_OVM7690_RegisterSet routine. Refer to the specific device data sheet for more information.

DRV_CAMERA_OVM7690_INDEX_0 Macro

OVM7690 driver index definitions

File

drv_camera_ovm7690.h

C

```
#define DRV_CAMERA_OVM7690_INDEX_0 0
```

Description

Driver Camera OVM7690 Module Index

These constants provide Camera OVM7690 driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_CAMERA_OVM7690_Initialize and DRV_CAMERA_OVM7690_Open routines to identify the driver instance in use.

DRV_CAMERA_OVM7690_INDEX_1 Macro

File

drv_camera_ovm7690.h

C

```
#define DRV_CAMERA_OVM7690_INDEX_1 1
```

Description

This is macro DRV_CAMERA_OVM7690_INDEX_1.

DRV_CAMERA_OVM7690_REG12_SOFT_RESET Macro

Driver Camera OVM7690 Register 0x12 Soft reset flag.

File

drv_camera_ovm7690.h

C

```
#define DRV_CAMERA_OVM7690_REG12_SOFT_RESET
```

Description

Driver Camera OVM7690 Soft reset flag.

This macro provide definition of Camera OVM7690 Register 0x12 Soft reset flag.

Remarks

These constants should be used in place of hard-coded numeric literals.

DRV_CAMERA_OVM7690_SCCB_READ_ID Macro

Camera OVM7690 SCCB Interface device Read Slave ID.

File

drv_camera_ovm7690.h

C

```
#define DRV_CAMERA_OVM7690_SCCB_READ_ID
```

Description

Driver Camera OVM7690 SCCB Read ID

This macro provide definition of Camera OVM7690 SCCB Interface device Read Slave ID.

Remarks

These constants should be used in place of hard-coded numeric literals.

DRV_CAMERA_OVM7690_SCCB_WRITE_ID Macro

Camera OVM7690 SCCB Interface device Write Slave ID

File

drv_camera_ovm7690.h

C

```
#define DRV_CAMERA_OVM7690_SCCB_WRITE_ID
```

Description

Driver Camera OVM7690 SCCB Write ID

This macro provide definition of Camera OVM7690 SCCB Interface device Write Slave ID.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_CAMERA_OVM7690_RegisterSet routine to identify the Camera OVM7690 SCCB Interface device Write Slave ID.

Files

Files

Name	Description
drv_camera_ovm7690.h	Camera OVM7690 Driver Local Data Structures
drv_ovm7690_config_template.h	OVM7690 Device Driver configuration template.

Description

drv_camera_ovm7690.h

Camera OVM7690 Driver Local Data Structures

Enumerations

	Name	Description
	DRV_CAMERA_OVM7690_CLIENT_STATUS	Identifies Camera OVM7690 possible client status.
	DRV_CAMERA_OVM7690_ERROR	Identifies Camera OVM7690 possible errors.
	DRV_CAMERA_OVM7690_REG12_OP_FORMAT	List of Camera OVM7690 Device Register Addresses.

Functions

	Name	Description
	DRV_CAMERA_OVM7690_Close	Closes an opened-instance of the Camera OVM7690 driver.
	DRV_CAMERA_OVM7690_Deinitialize	Deinitializes the specified instance of the Camera OVM7690 driver module.
	DRV_CAMERA_OVM7690_FrameBufferAddressSet	Sets Frame buffer address.
	DRV_CAMERA_OVM7690_FrameRectSet	Sets the Frame Rectangle Set.
	DRV_CAMERA_OVM7690_HsyncEventHandler	Horizontal Synchronization Event Handler
	DRV_CAMERA_OVM7690_Initialize	Initializes the Camera OVM7690 instance for the specified driver index.
	DRV_CAMERA_OVM7690_Open	Opens the specified Camera OVM7690 driver instance and returns a handle to it.
	DRV_CAMERA_OVM7690_RegisterSet	Sets the camera OVM7690 configuration registers
	DRV_CAMERA_OVM7690_Start	Starts camera rendering to the display.
	DRV_CAMERA_OVM7690_Stop	Stops rendering the camera Pixel data.
	DRV_CAMERA_OVM7690_VsyncEventHandler	Vertical Synchronization Event Handler

Macros

	Name	Description
	DRV_CAMERA_OVM7690_INDEX_0	OVM7690 driver index definitions
	DRV_CAMERA_OVM7690_INDEX_1	This is macro DRV_CAMERA_OVM7690_INDEX_1.
	DRV_CAMERA_OVM7690_REG12_SOFT_RESET	Driver Camera OVM7690 Register 0x12 Soft reset flag.
	DRV_CAMERA_OVM7690_SCCB_READ_ID	Camera OVM7690 SCCB Interface device Read Slave ID.
	DRV_CAMERA_OVM7690_SCCB_WRITE_ID	Camera OVM7690 SCCB Interface device Write Slave ID

Structures

	Name	Description
	DRV_CAMERA_OVM7690_CLIENT_OBJ	Camera OVM7690 Driver Client Object.
	DRV_CAMERA_OVM7690_INIT	Camera OVM7690 Driver Initialization Parameters.
	DRV_CAMERA_OVM7690_OBJ	Camera OVM7690 Driver Instance Object.
	DRV_CAMERA_OVM7690_RECT	Camera OVM7690 Window Rectangle co-ordinates.

Description

Camera OVM7690 Driver Local Data Structures

Driver Local Data Structures

File Name

drv_camera_ovm7690.h

Company

Microchip Technology Inc.

drv_ovm7690_config_template.h

OVM7690 Device Driver configuration template.

Macros

	Name	Description
	DRV_OVM7690_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.

Description

OVM7690 Device Driver Configuration Template

This header file contains the build-time configuration selections for the OVM7690 device driver. This is the template file which give all possible configurations that can be made. This file should not be included in any project.

File Name

drv_ovm7690_config_template.h

Company

Microchip Technology Inc.

CAN Driver Library

This topic describes the CAN Driver Library.

Introduction

The CAN Static Driver provides a high-level interface to manage the CAN module on the Microchip family of microcontrollers.

Description

Through MHC, this driver provides an API to initialize the CAN module, as well as baud rate. The API also allows simple transmit and receive functionality.

Library Interface

Function(s)

	Name	Description
	DRV_CAN_ChannelMessageReceive	Receives a message on a channel for the specified driver index. **Implementation:** Static
	DRV_CAN_ChannelMessageTransmit	Transmits a message on a channel for the specified driver index. **Implementation:** Static
	DRV_CAN_Close	Closes the CAN instance for the specified driver index. **Implementation:** Static
	DRV_CAN_Deinitialize	Deinitializes the DRV_CAN_Initialize instance that has been called for the specified driver index. **Implementation:** Static
	DRV_CAN_Initialize	Initializes the CAN instance for the specified driver index. **Implementation:** Static
	DRV_CAN_Open	Opens the CAN instance for the specified driver index. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the CAN Driver Library.

Function(s)

DRV_CAN_ChannelMessageReceive Function

Receives a message on a channel for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
bool DRV_CAN_ChannelMessageReceive(CAN_CHANNEL channelNum, int address, uint8_t DLC, uint8_t*
message);
```

Returns

Boolean "true" when a message has been received.

Description

This routine receives data into a buffer from the CAN bus according to the channel, address, and data length given.

Remarks

This routine receives a standard or extended messages based upon the CAN Driver setup.

Preconditions

DRV_CAN_Initialize has been called.

Parameters

Parameters	Description
CAN_CHANNEL channelNum	CAN channel to use
int address	CAN address to receive on
uint8_t DLC	Data Length Code of Message
uint8_t* message	Pointer to put the message data to receive

Function

bool DRV_CAN_ChannelMessageReceive(CAN_CHANNEL channelNum, int address, uint8_t DLC, uint8_t* message);

DRV_CAN_ChannelMessageTransmit Function

Transmits a message on a channel for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
bool DRV_CAN_ChannelMessageTransmit(CAN_CHANNEL channelNum, int address, uint8_t DLC, uint8_t*
message);
```

Returns

Boolean "true" when a message has been transmitted.

Description

This routine transmits a data buffer on the CAN bus according to the channel, address, and data length given.

Remarks

This routine receives a standard or extended messages based upon the CAN Driver setup.

Preconditions

DRV_CAN_Initialize has been called.

Parameters

Parameters	Description
CAN_CHANNEL channelNum	CAN channel to use
int address	CAN address to transmit on
uint8_t DLC	Data Length Code of Message
uint8_t* message	Pointer to the message data to send

Function

bool DRV_CAN_ChannelMessageTransmit(CAN_CHANNEL channelNum, int address, uint8_t DLC, uint8_t*
message);

DRV_CAN_Close Function

Closes the CAN instance for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
void DRV_CAN_Close();
```

Returns

None.

Description

This routine closes the CAN driver instance for the specified driver instance, making it ready for clients to use it.

Preconditions

DRV_CAN_Initialize has been called.

Function

void DRV_CAN_Close(void)

DRV_CAN_Deinitialize Function

Deinitializes the DRV_CAN_Initialize instance that has been called for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
void DRV_CAN_Deinitialize();
```

Returns

None.

Description

This routine deinitializes the CAN Driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Preconditions

None.

Function

void DRV_CAN_Deinitialize(void)

DRV_CAN_Initialize Function

Initializes the CAN instance for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
void DRV_CAN_Initialize();
```

Returns

None.

Description

This routine initializes the CAN Driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

This routine must be called before any other CAN routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_CAN_Initialize(void)

DRV_CAN_Open Function

Opens the CAN instance for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
void DRV_CAN_Open();
```

Returns

None.

Description

This routine opens the CAN Driver instance for the specified driver instance, making it ready for clients to use it.

Preconditions

DRV_CAN_Initialize has been called.

Function

void DRV_CAN_Open(void)

Codec Driver Libraries

This section describes the Codec Driver Libraries available in MPLAB Harmony.

AK4384 Codec Driver Library

This topic describes the AK4384 Codec Driver Library.

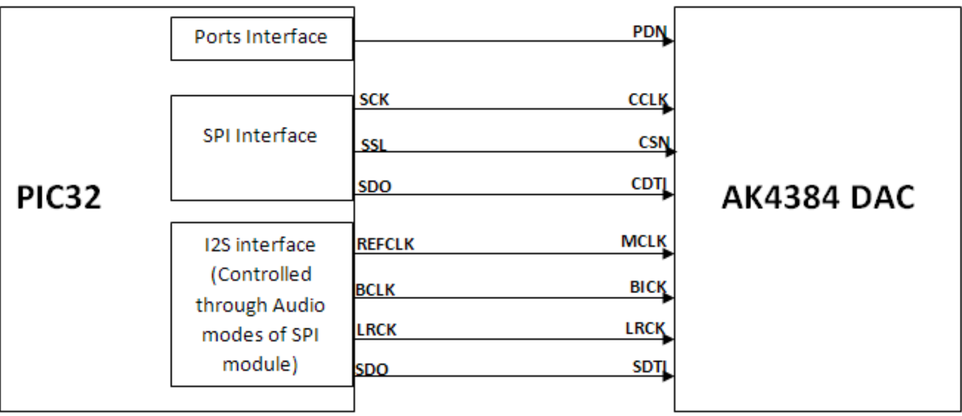
Introduction

This library provides an interface to manage the AK4384 106 dB 192 kHz 24-Bit DAC that is serially interfaced to a Microchip microcontroller for providing Audio Solutions.

Description

The AK4384 module is 24-bit Audio DAC from Asahi Kasei Microdevices Corporation. The AK4384 can be interfaced to Microchip microcontrollers through SPI and I2S serial interfaces. SPI interface is used for control command transfer. The I2S interface is used for Audio data output.

A typical interface of AK4384 to a Microchip PIC32 device is provided in the following diagram:



Features

The AK4384 Codec Driver supports the following features:

- Sampling Rate Ranging from 8 kHz to 192 kHz
- 128 times Oversampling (Normal Speed mode)
- 64 times Oversampling (Double Speed mode)
- 32 times Oversampling (Quad Speed mode)
- Digital de-emphasis for 32k, 44.1k and 48 kHz sampling
- Soft mute
- Digital Attenuator (Linear 256 steps)
- I/F format:
  - 24-bit MSB justified
  - 24/20/16-bit LSB justified
  - I2S
- Master clock:
  - 256 fs, 384 fs, 512 fs, 768 fs, or 1152 fs (Normal Speed mode)
  - 128 fs, 192 fs, 256 fs, or 384 fs (Double Speed mode)
  - 128 fs or 192 fs (Quad Speed mode)

Using the Library

This topic describes the basic architecture of the AK4384 Codec Driver Library and provides information and

examples on its use.

Description

Interface Header File: `drv_ak4384.h`

The interface to the AK4384 Codec Driver library is defined in the `drv_ak4384.h` header file. Any C language source (`.c`) file that uses the AK4384 Codec Driver library should include this header.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

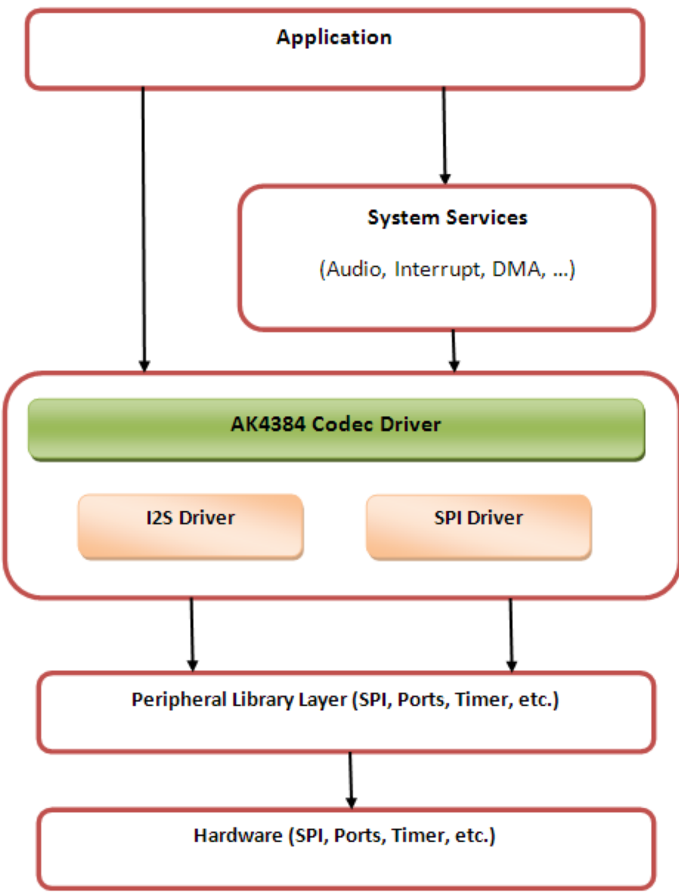
Abstraction Model

This library provides a low-level abstraction of the AK4384 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the AK4384 Codec Driver is positioned in the MPLAB Harmony framework. The AK4384 Codec Driver uses the SPI and I2S drivers for control and audio data transfers to the AK4384 module.

AK4384 Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The AK4384 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK4384 DAC module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK4384 Codec Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open and close functions.
Codec Specific Functions	Provides functions that are codec specific.
Data Transfer Functions	Provides data transfer functions.
Other Functions	Provides driver specific miscellaneous functions such as sampling rate setting, control command functions, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the AK4384 Codec Driver Library.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

System Access

This topic provides information on system initialization, implementations, and provides a system access code example.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the AK4384 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK4384_INIT or by using Initialization Overrides) that are supported by the specific AK4384 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- SPI driver module index. The module index should be same as the one used in initializing the SPI Driver.
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver.
- Sampling rate
- Master clock detection mode
- Power down pin port initialization
- Queue size for the audio data transmit buffer

The DRV_AK4384_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ AK4384_Deinitialize, DRV_ AK4384_Status and DRV_I2S_Tasks.

Implementations

The AK4384 Codec Driver can have the following implementations:

Implementation	Description	MPLAB Harmony Components
Implementation 1	Dedicated hardware for control (SPI) and data (I2S) interface.	Standard MPLAB Harmony drivers for SPI and I2S interfaces.
Implementation 2	Dedicated hardware for data (I2S) interface. Ports pins for control interface.	Standard MPLAB Harmony drivers for I2S interface. Virtual MPLAB Harmony drivers for SPI interface.
Implementation 3	Dedicated hardware for data (I2S) interface. Ports pins for control.	Standard MPLAB Harmony drivers for I2S interface. An internal bit-banged implementation of control interface in the AK4384 Codec Driver.

If Implementation 3 is in use, while initializing fields of DRV_AK4384_INIT structure, the SPI Driver module index initialization is redundant. The user can pass a dummy value.

For Implementation 3, the user has to additionally initialize parameters to support bit-banged control interface implementation. These additional parameters can be passed by assigning values to the respective macros in `system_config.h`.

Example:

```
DRV_AK4384_INIT drvak4384Init =
{
```

```
        .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
        .volume = 120,
        .mclkMode = DRV_AK4384_MCLK_MODE_MANUAL,
        .queueSizeTransmit = 2,
};

/*
 The SPI module index should be same as the one used in
 initializing the SPI driver.
 The SPI module index initialization is redundant
 if Implementation 3 is in use.
 */
drvak4384Init.spiDriverModuleIndex = DRV_SPI_INDEX_0;

/*
 The I2S module index should be same as the one used in
 initializing the I2S driver.
 */
drvak4384Init.i2sDriverModuleIndex = DRV_I2S_INDEX_0;

ak4384DevObject = DRV_AK4384_Initialize(DRV_AK4384_INDEX_0, (SYS_MODULE_INIT *) &drvak4384Init);
if (SYS_MODULE_OBJ_INVALID == ak4384DevObject)
{
    // Handle error
}
```

Task Routine

The DRV_AK4384_Tasks will be called from the System Task Service.

Client Access

This topic describes client access and includes a code example.

Description

For the application to start using an instance of the module, it must call the DRV_AK4384_Open function. The DRV_AK4384_Open provides a driver handle to the AK4384 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK4384_Deinitialize, the application must call the DRV_AK4384_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Note: It is necessary to check the status of driver initialization before opening a driver instance. The status of the AK4384 Codec Driver can be known by calling DRV_AK4384_Status.

Example:

```
DRV_HANDLE handle;
SYS_STATUS ak4384Status;
    ak4384Status = DRV_AK4384_Status(sysObjects.ak4384DevObject);
        if (SYS_STATUS_READY == ak4384Status)
        {
               // The driver can now be opened.
               appData.ak4384Client.handle = DRV_AK4384_Open
                                             (DRV_AK4384_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
               if(appData.ak4384Client.handle != DRV_HANDLE_INVALID)
               {
                   appData.state = APP_STATE_AK4384_SET_BUFFER_HANDLER;
               }
               else
               {
                   SYS_DEBUG(0, "Find out what's wrong \r\n");
               }
    }
```

```
    else
    {
        /* AK4384 Driver Is not ready */
                ;
    }
```

Client Operations

This topic describes client operations and provides a code example.

Description

1. The calling and execution of the below functions does not guarantee that the function (and its associated CODEC command) has been set in the CODEC peer interfaced through SPI. It just means that the submission of the command has started over the SPI interface.

2. In regard to the Note 1 above, the user should not call the below functions consecutively which could result in unexpected behavior. If needed the user should confirm the completion status of a function before calling any of the other functions.

3. To know the completion status of the below functions user can register a command event callback handler by calling 'DRV_AK4384_CommandEventHandlerSet'. The callback handler will be called when the last submitted command (submitted by calling one of the below functions) has completed.

Notes:

1. The calling and execution of the following functions does not guarantee that the function (and its associated Codec command) has been set in the Codec peer interfaced through the SPI. It just means that the submission of the command has started over the SPI.
2. Regarding Note 1, the user should not call the following functions consecutively, which could result in unexpected behavior. If needed, the user should confirm the completion status of a function before calling any of the other functions.
3. To know the completion status of the following functions, users can register a command event callback handler by calling the function 'DRV_AK4384_CommandEventHandlerSet'. The callback handler will be called when the last submitted command (submitted by calling one of the following functions) has completed.

Client operations provide the API interface for control command and audio data transfer to the AK4384 DAC.

The following AK4384 DAC specific control command functions are provided:

- DRV_AK4384_SamplingRateSet
- DRV_AK4384_SamplingRateGet
- DRV_AK4384_VolumeSet
- DRV_AK4384_VolumeGet
- DRV_AK4384_MuteOn
- DRV_AK4384_MuteOff
- DRV_AK4384_ZeroDetectEnable
- DRV_AK4384_ZeroDetectDisable
- DRV_AK4384_ZeroDetectModeSet
- DRV_AK4384_ZeroDetectInvertEnable
- DRV_AK4384_ZeroDetectInvertDisable
- DRV_AK4384_ChannelOutputInvertEnable
- DRV_AK4384_ChannelOutputInvertDisable
- DRV_AK4384_SlowRollOffFilterEnable
- DRV_AK4384_SlowRollOffFilterDisable
- DRV_AK4384_DeEmphasisFilterSet

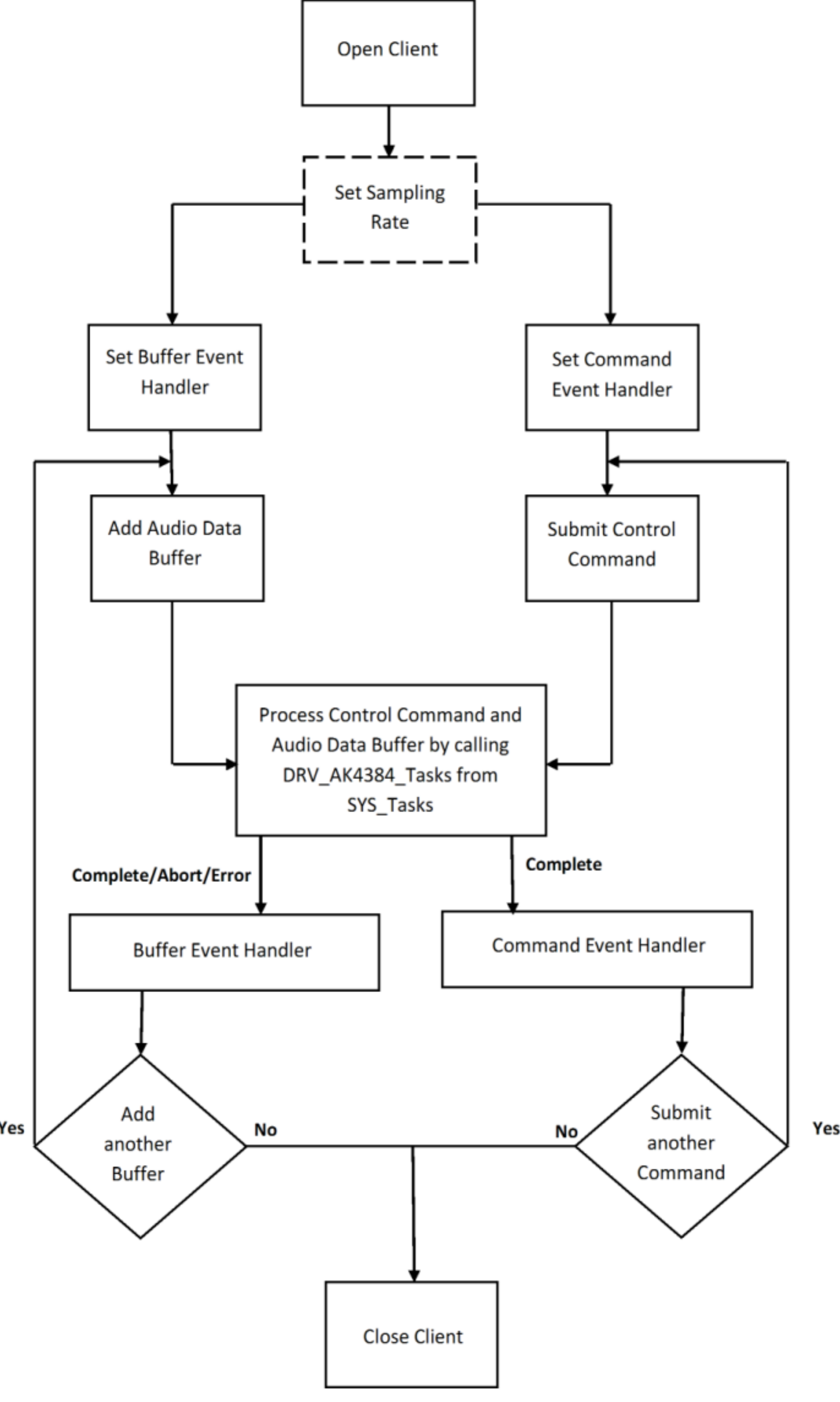
These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the AK4384 DAC. A notification for the submitted requests can be received by registering a command callback event with the driver. The driver notifies by calling the callback on successfully transmitting the

command to the AK4384 DAC module.

The function DRV_AK4384_BufferAddWrite is a buffered data operation functions. This function schedules non-blocking audio data transfer operation. The function adds the request to the hardware instance queues and returns a buffer handle. The requesting client also registers a callback event with the driver. The driver notifies the client with DRV_AK4384_BUFFER_EVENT_COMPLETE, DRV_AK4384_BUFFER_EVENT_ERROR, or DRV_AK4384_BUFFER_EVENT_ABORT events.

The submitted control commands and audio buffer add requests are processed under DRV_AK4384_Tasks function. This function is called from the SYS_Tasks routine.

The following diagram illustrates the control commands and audio buffered data operations.



Note: It is not necessary to close and reopen the client between multiple transfers.

An application using the buffered functionality needs to perform the following steps:

1. The system should have completed necessary setup and initializations.

2. The I2S driver object should have been initialized by calling DRV_I2S_Initialize.
3. The SPI driver object should have been initialized by calling DRV_SPI_Initialize.
4. The AK4384 driver object should be initialized by calling DRV_AK4384_Initialize.
5. The necessary sampling rate value should be set up by calling DRV_AK4384_ SamplingRateSet.
6. Register buffer event handler for the client handle by calling DRV_AK4384_BufferEventHandlerSet.
7. Register command event handler for the client handle by calling DRV_AK4384_CommandEventHandlerSet.
8. Submit a command by calling specific command API.
9. Add a buffer to initiate the data transfer by calling DRV_AK4384_BufferAddWrite.
10. The submitted command and Audio data processing happens b calling DRV_AK4384_Tasks from SYS_Tasks.
11. Repeat steps 9 through 10 to handle multiple buffer transmission and reception.
12. When the client is done, it can use DRV_AK4384_Close to close the client handle.

Example:

```
typedef enum
{
    APP_STATE_AK4384_OPEN,
    APP_STATE_AK4384_SET_COMMAND_HANDLER,
    APP_STATE_AK4384_SET_BUFFER_HANDLER,
    APP_STATE_AK4384_SET_SAMPLING_RATE_COMMAND,
    APP_STATE_AK4384_ADD_BUFFER,
    APP_STATE_AK4384_WAIT_FOR_BUFFER_COMPLETE,
    APP_STATE_AK4384_BUFFER_COMPLETE
} APP_STATES;

typedef struct
{
    DRV_HANDLE handle;
    DRV_AK4384_BUFFER_HANDLE writeBufHandle;
    DRV_AK4384_BUFFER_EVENT_HANDLER bufferHandler;
    DRV_AK4384_COMMAND_EVENT_HANDLER commandHandler;
    uintptr_t context;
    uint8_t *txbufferObject;
    size_t bufferSize;

} APP_AK4384_CLIENT;

typedef struct
{
    /* Application's current state*/
    APP_STATES state;
    /* USART client handle */
    APP_AK4384_CLIENT ak4384Client;
} APP_DATA;
APP_DATA appData;
SYS_MODULE_OBJ ak4384DevObject;
DRV_AK4384_INIT drvak4384Init =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .volume = 120,
    .mclkMode = DRV_AK4384_MCLK_MODE_MANUAL,
    .queueSizeTransmit = 2,
};

void SYS_Initialize(void * data)
{
    /*
     The SPI module index should be same as the one used in
     initializing the SPI driver.
     The SPI module index initialization is redundant
     if Implementation 3 (Described in System Access) is in use.
     */
    drvak4384Init.spiDriverModuleIndex = DRV_SPI_INDEX_0;

    /*
     The I2S module index should be same as the one used in
```

```
     initializing the I2S driver.
     */
    drvak4384Init.i2sDriverModuleIndex = DRV_I2S_INDEX_0;

    ak4384DevObject = DRV_AK4384_Initialize(DRV_AK4384_INDEX_0, (SYS_MODULE_INIT *) & drvak4384Init);
    if (SYS_MODULE_OBJ_INVALID == ak4384DevObject) {
        // Handle error
    }
}

void APP_Tasks (void )
{
    switch(appData.state)
    {
        /* Open the ak4384 client and get an Handle */
        case APP_STATE_AK4384_OPEN:
        {
            SYS_STATUS ak4384Status;
            ak4384Status = DRV_AK4384_Status(sysObjects.ak4384DevObject);
            if (SYS_STATUS_READY == ak4384Status)
            {
                // This means the driver can now be opened.
                appData.ak4384Client.handle = DRV_AK4384_Open(DRV_AK4384_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
                if(appData.ak4384Client.handle != DRV_HANDLE_INVALID)
                {
                    appData.state = APP_STATE_AK4384_SET_COMMAND_HANDLER;
                }
                else
                {
                    SYS_DEBUG(0, "Find out what is wrong \r\n");
                }
            }
            else
            {
                /* Wait for AK4384 to Initialize */
                ;
            }
        }
        break;

        /* Register a command event handler */
        case APP_STATE_AK4384_SET_COMMAND_HANDLER:
        {
            DRV_AK4384_CommandEventHandlerSet(appData.ak4384Client.handle,
                    appData.ak4384Client.commandHandler,
                    appData.ak4384Client.context);
            appData.state = APP_STATE_AK4384_SET_BUFFER_HANDLER;
        }
        break;


        /* Register a buffer event handler */
        case APP_STATE_AK4384_SET_BUFFER_HANDLER:
        {
            DRV_AK4384_BufferEventHandlerSet(appData.ak4384Client.handle,
                    appData.ak4384Client.bufferHandler,
                    appData.ak4384Client.context);
            appData.state = APP_STATE_AK4384_SET_SAMPLING_RATE_COMMAND;
        }
        break;

        /* Submit a set sampling rate command */
        case APP_STATE_AK4384_SET_SAMPLING_RATE_COMMAND:
        {
            DRV_AK4384_SamplingRateSet(appData.ak4384Client.handle,48000);
            appData.state = APP_STATE_AK4384_ADD_BUFFER;
        }
        break;
```

```
        /* Add the Audio buffer to be transmitted */
        case APP_STATE_AK4384_ADD_BUFFER:
        {
            DRV_AK4384_BufferAddWrite(appData.ak4384Client.handle, &appData.ak4384Client.writeBufHandle,
            appData.ak4384Client.txbufferObject, appData.ak4384Client.bufferSize);
            if(appData.ak4384Client.writeBufHandle != DRV_AK4384_BUFFER_HANDLE_INVALID)
            {
                appData.state = APP_STATE_AK4384_WAIT_FOR_BUFFER_COMPLETE;
            }
            else
            {
                SYS_DEBUG(0, "Find out what is wrong \r\n");
            }
        }
        break;

    /* Audio Buffer transmission under process */
        case APP_STATE_AK4384_WAIT_FOR_BUFFER_COMPLETE:
        {
        }
        break;

    /* Audio Buffer transmission completed */
        case APP_STATE_AK4384_BUFFER_COMPLETE:
        {
            /* Add another buffer */
            appData.state = APP_STATE_AK4384_ADD_BUFFER;
        }
        break;

        default:
        {
        }
        break;
    }

}

void APP_AK4384CommandEventHandler(uintptr_t context )
{

    // Last submitted command successful. Take action as needed.
}

void APP_AK4384BufferEventHandler(DRV_AK4384_BUFFER_EVENT event,
        DRV_AK4384_BUFFER_HANDLE handle, uintptr_t context )
{

    switch(event)
    {
        case DRV_AK4384_BUFFER_EVENT_COMPLETE:
        {
            // Can set appData.state = APP_STATE_AK4384_BUFFER_COMPLETE;
        // Take Action as needed

        }
        break;
        case DRV_AK4384_BUFFER_EVENT_ERROR:
        {
        // Take Action as needed

        } break;

        case DRV_AK4384_BUFFER_EVENT_ABORT:
        {
        // Take Action as needed

        } break;
```

```
    }
}

void SYS_Tasks(void)
{
    DRV_AK4384_Tasks(ak4384DevObject);
    APP_Tasks();
}
```

Configuring the Library

Macros

	Name	Description
	DRV_AK4384_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4384_CONTROL_CLOCK	Sets up clock frequency for the control interface (SPI)
	DRV_AK4384_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4384_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4384_TIMER_DRIVER_MODULE_INDEX	Identifies the Timer Module Index for custom virtual SPI driver implementation.
	DRV_AK4384_TIMER_PERIOD	Identifies the period for the bit bang timer.
	DRV_AK4384_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1, and 48K sampling frequency
	DRV_AK4384_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1 and 48K sampling frequency

Description

The configuration of the AK4384 Codec Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the AK4384 Codec Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the AK4384 Codec Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

DRV_AK4384_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_CLIENTS_NUMBER DRV_AK4384_INSTANCES_NUMBER
```

Description

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances. Therefore, if there are five AK4384 hardware interfaces, this number will be 5.

Remarks

None.

Section

Client Configuration

****

****

****

AK4384 Client Count Configuration

DRV_AK4384_CONTROL_CLOCK Macro

Sets up clock frequency for the control interface (SPI)

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_CONTROL_CLOCK
```

Description

AK4384 Control Interface Clock Speed configuration

Sets up clock frequency for the control interface (SPI). The maximum value supported is 5MHZ.

Remarks

1. This Macro is useful only when a hardware SPI module is not available(used) or a virtual SPI driver is not available(used) for the control interface to the AK4384 CODEC.
2. This constant needs to defined only for a bit banged implementation of control interface with in the driver.

DRV_AK4384_INPUT_REFCLOCK Macro

Identifies the input REFCLOCK source to generate the MCLK to codec.

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_INPUT_REFCLOCK
```

Description

AK4384 Input reference clock

Identifies the input REFCLOCK source to generate the MCLK to codec.

Remarks

None.

DRV_AK4384_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_INSTANCES_NUMBER
```

Description

AK4384 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of AK4384 CODEC modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_AK4384_TIMER_DRIVER_MODULE_INDEX Macro

Identifies the Timer Module Index for custom virtual SPI driver implementation.

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_TIMER_DRIVER_MODULE_INDEX
```

Description

AK4384 Timer Module Index

Identifies the Timer Module Index for custom virtual SPI driver implementation. The AK4384 uses SPI protocol for control interface. The Timer Module Index is needed by AK4384 driver to implement a virtual SPI driver for control command exchange with the AK4384 CODEC.

Remarks

1. This Macro is useful only when a hardware SPI module is not available(used) or a virtual SPI driver is not available(used) for the control interface to the AK4384 CODEC.
2. This constant needs to defined only for a bit banged implementation of control interface with in the driver.

DRV_AK4384_TIMER_PERIOD Macro

Identifies the period for the bit bang timer.

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_TIMER_PERIOD
```

Description

AK4384 Timer Period

Identifies the period for the bit bang timer after which the timer interrupt should occur. The value assigned should align with the expected control interface clock defined by AK4384_CONTROL_CLOCK.

Remarks

1. This Macro is useful only when a hardware SPI module is not available(used) or a virtual SPI driver is not available(used) for the control interface to the AK4384 CODEC.
2. This constant needs to defined only for a bit banged implementation of control interface with in the driver.

DRV_AK4384_BCLK_BIT_CLK_DIVISOR Macro

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1, and 48K sampling frequency

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_BCLK_BIT_CLK_DIVISOR
```

Description

AK4384 BCLK to LRCK Ratio to Generate Audio Stream

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1 and 48K I2S sampling frequency

Following BCLK to LRCK ratios are supported 16bit LSB Justified >=32fs 20bit LSB Justified >=40fs 24bit MSB Justified >=48fs 24bit I2S Compatible >=48fs 24bit LSB Justified >=48fs

Typical values for the divisor are 1,2,4 and 8

Remarks

None.

DRV_AK4384_MCLK_SAMPLE_FREQ_MULTPLIER Macro

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1 and 48K sampling frequency

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_MCLK_SAMPLE_FREQ_MULTPLIER
```

Description

AK4384 MCLK to LRCK Ratio to Generate Audio Stream

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1, and 48K I2S sampling frequency

Supported MCLK to LRCK Ratios are as below 256fs, 384fs, 512fs, 768fs or 1152fs [Normal Speed Mode(8kHz~48kHz)] 128fs, 192fs, 256fs or 384fs [Double Speed Mode(60kHz~96kHz)] 128fs, 192fs [Quad Speed Mode(120kHz~192kHz)]

Remarks

None

Building the Library

This section lists the files that are available in the AK4384 Codec Driver Library.

Description

This section list the files that are available in the `/src` folder of the AK4384 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/codec/ak4384`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_ak4384.h	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_ak4384_bit_banged_control_interface.c	This file contains implementation of the AK4384 Codec Driver with a custom bit-banged implementation for control interface driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired

implementation.

Source File Name	Description
/src/dynamic/drv_ak4384_virtual_control_interface.c	This file contains implementation of the AK4384 Codec Driver with a virtual SPI driver as control interface driver. **Note:** This file is currently unsupported.
/src/dynamic/drv_ak4384.c	This file contains the core implementation of the AK4384 Codec Driver **Note:** This file currently unsupported.

Module Dependencies

The AK4384 Driver Library depends on the following modules:

- I2S Driver Library
- SPI Driver Library
- Timer Driver Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_AK4384_Initialize	Initializes hardware and data for the instance of the AK4384 DAC module. **Implementation:** Dynamic
	DRV_AK4384_Deinitialize	Deinitializes the specified instance of the AK4384 driver module. **Implementation:** Dynamic
	DRV_AK4384_Status	Gets the current status of the AK4384 driver module. **Implementation:** Dynamic
	DRV_AK4384_Tasks	Maintains the driver's control and data interface state machine. **Implementation:** Dynamic

b) Client Setup Functions

	Name	Description
	DRV_AK4384_Open	Opens the specified AK4384 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_AK4384_Close	Closes an opened-instance of the AK4384 driver. **Implementation:** Dynamic

c) Codec Specific Functions

	Name	Description
	DRV_AK4384_ChannelOutputInvertDisable	Disables output polarity of the selected Channel. **Implementation:** Dynamic
	DRV_AK4384_ChannelOutputInvertEnable	Enables output polarity of the selected channel. **Implementation:** Dynamic
	DRV_AK4384_DeEmphasisFilterSet	Allows specifies enabling of digital de-emphasis filter. **Implementation:** Dynamic
	DRV_AK4384_MuteOff	Disables AK4384 output for soft mute. **Implementation:** Dynamic

	DRV_AK4384_MuteOn	Allows AK4384 output for soft mute on. **Implementation:** Dynamic
	DRV_AK4384_SamplingRateGet	This function gets the sampling rate set on the DAC AK4384. **Implementation:** Dynamic
	DRV_AK4384_SamplingRateSet	This function sets the sampling rate of the media stream. **Implementation:** Dynamic
	DRV_AK4384_SlowRollOffFilterDisable	Disables Slow Roll-off filter function. **Implementation:** Dynamic
	DRV_AK4384_SlowRollOffFilterEnable	Enables Slow Roll-off filter function. **Implementation:** Dynamic
	DRV_AK4384_VolumeGet	This function gets the volume for AK4384 Codec. **Implementation:** Dynamic
	DRV_AK4384_VolumeSet	This function sets the volume for AK4384 Codec. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectDisable	Disables AK4384 channel-independent zeros detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectEnable	Enables AK4384 channel-independent zeros detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectInvertDisable	Disables inversion of polarity for zero detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectInvertEnable	Enables inversion of polarity for zero detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectModeSet	Sets mode of AK4384 channel-independent zeros detect function. **Implementation:** Dynamic

d) Data Transfer Functions

	Name	Description
	DRV_AK4384_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_AK4384_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_AK4384_BufferCombinedQueueSizeGet	This function returns the number of bytes queued (to be processed) in the buffer queue. **Implementation:** Dynamic
	DRV_AK4384_BufferProcessedSizeGet	This function returns number of bytes that have been processed for the specified buffer. **Implementation:** Dynamic

e) Other Functions

	Name	Description
	DRV_AK4384_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. **Implementation:** Dynamic

	DRV_AK4384_VersionGet	Returns the version of the AK4384 driver. **Implementation:** Dynamic
	DRV_AK4384_VersionStrGet	Returns the version of AK4384 driver in string format. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	DRV_AK4384_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4384_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4384_BUFFER_EVENT_HANDLER	Pointer to a AK4384 Driver Buffer Event handler function.
	DRV_AK4384_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4384_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4384_COMMAND_EVENT_HANDLER	Pointer to a AK4384 Driver Command Event Handler Function
	DRV_AK4384_DEEMPHASIS_FILTER	Identifies de-emphasis filter function.
	DRV_AK4384_INIT	Defines the data required to initialize or reinitialize the AK4384 driver.
	DRV_AK4384_MCLK_MODE	Identifies the mode of master clock to AK4384 DAC.
	DRV_AK4384_ZERO_DETECT_MODE	Identifies Zero Detect Function mode
	DRV_AK4384_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4384_COUNT	Number of valid AK4384 driver indices.
	DRV_AK4384_INDEX_0	AK4384 driver index definitions.
	DRV_AK4384_INDEX_1	This is macro DRV_AK4384_INDEX_1.
	DRV_AK4384_INDEX_2	This is macro DRV_AK4384_INDEX_2.
	DRV_AK4384_INDEX_3	This is macro DRV_AK4384_INDEX_3.
	DRV_AK4384_INDEX_4	This is macro DRV_AK4384_INDEX_4.
	DRV_AK4384_INDEX_5	This is macro DRV_AK4384_INDEX_5.

Description

This section describes the API functions of the AK4384 Codec Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_AK4384_Initialize Function

Initializes hardware and data for the instance of the AK4384 DAC module.

Implementation: Dynamic

File

drv_ak4384.h

C

```
SYS_MODULE_OBJ DRV_AK4384_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the AK4384 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the 'init' parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Remarks

This routine must be called before any other AK4384 routine is called.

This routine should only be called once during system initialization unless DRV_AK4384_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this CODEC driver. DRV_SPI_Initialize must be called if SPI driver is used for handling the control interface of this CODEC driver.

Example

```
DRV_AK4384_INIT                 init;
SYS_MODULE_OBJ                  objectHandle;

init.moduleInit.value           = SYS_MODULE_POWER_RUN_FULL;
init.spiDriverModuleIndex       = DRV_SPI_INDEX_0;  // This will be ignored for a custom
                                                    // control interface driver implementation
init.i2sDriverModuleIndex       = DRV_I2S_INDEX_0;
init.mclkMode                   = DRV_AK4384_MCLK_MODE_MANUAL;
init.audioDataFormat            = DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_I2S;
init.powerDownPortChannel       = PORT_CHANNEL_G;
init.powerDownBitPosition       = PORTS_BIT_POS_15;

objectHandle = DRV_AK4384_Initialize(DRV_AK4384_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_AK4384_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT *const init

);

DRV_AK4384_Deinitialize Function

Deinitializes the specified instance of the AK4384 driver module.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the AK4384 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_AK4384_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_AK4384_Initialize
SYS_STATUS          status;


DRV_AK4384_Deinitialize(object);

status = DRV_AK4384_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4384_Initialize routine

Function

void DRV_AK4384_Deinitialize(SYS_MODULE_OBJ object)

DRV_AK4384_Status Function

Gets the current status of the AK4384 driver module.

Implementation: Dynamic

File

drv_ak4384.h

C

```
SYS_STATUS DRV_AK4384_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This routine provides the current status of the AK4384 driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_AK4384_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4384_Initialize
SYS_STATUS          ak4384Status;

ak4384Status = DRV_AK4384_Status(object);
if (SYS_STATUS_READY == ak4384Status)
{
    // This means the driver can be opened using the
    // DRV_AK4384_Open function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4384_Initialize routine

Function

SYS_STATUS DRV_AK4384_Status(SYS_MODULE_OBJ object)

DRV_AK4384_Tasks Function

Maintains the driver's control and data interface state machine.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks function.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4384_Initialize

while (true)
{
    DRV_AK4384_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4384_Initialize)

Function

void DRV_AK4384_Tasks(SYS_MODULE_OBJ object);

b) Client Setup Functions

DRV_AK4384_Open Function

Opens the specified AK4384 driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_ak4384.h

C

```
DRV_HANDLE DRV_AK4384_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Errors can occur under following conditions:

- if the number of client objects allocated via DRV_AK4384_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid
- if the ioIntent options passed are not relevant to this driver

Description

This routine opens the specified AK4384 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

Only DRV_IO_INTENT_WRITE is a valid ioIntent option as AK4384 is DAC only.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_AK4384_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

Function DRV_AK4384_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_AK4384_Open(DRV_AK4384_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

```
DRV_HANDLE DRV_AK4384_Open
(
const SYS_MODULE_INDEX drvIndex,
const      DRV_IO_INTENT ioIntent
)
```

DRV_AK4384_Close Function

Closes an opened-instance of the AK4384 driver.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This routine closes an opened-instance of the AK4384 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_AK4384_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_AK4384_Open

DRV_AK4384_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

```
void DRV_AK4384_Close( DRV_Handle handle )
```

c) Codec Specific Functions

DRV_AK4384_ChannelOutputInvertDisable Function

Disables output polarity of the selected Channel.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ChannelOutputInvertDisable(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan);
```

Returns

None.

Description

This function disables output polarity of the selected Channel.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ChannelOutputInvertDisable(myAK4384Handle, DRV_AK4384_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Left or Right channel

Function

void DRV_AK4384_ChannelOutputInvertDisable(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan)

DRV_AK4384_ChannelOutputInvertEnable Function

Enables output polarity of the selected channel.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ChannelOutputInvertEnable(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan);
```

Returns

None.

Description

This function enables output polarity of the selected channel.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ChannelOutputInvertEnable(myAK4384Handle, DRV_AK4384_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Left or Right channel

Function

void DRV_AK4384_ChannelOutputInvertEnable(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan)

DRV_AK4384_DeEmphasisFilterSet Function

Allows specifies enabling of digital de-emphasis filter.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_DeEmphasisFilterSet(DRV_HANDLE handle, DRV_AK4384_DEEMPHASIS_FILTER filter);
```

Returns

None.

Description

This function allows specifies enabling of digital de-emphasis for 32, 44.1 or 48 kHz sampling rates (tc = 50/15 µs)

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_DeEmphasisFilterSet(myAK4384Handle, DRV_AK4384_DEEMPHASIS_FILTER_44_1KHZ)
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
filter	Specifies Enable of de-emphasis filter

Function

void DRV_AK4384_DeEmphasisFilterSet
(
    DRV_HANDLE handle,
    DRV_AK4384_DEEMPHASIS_FILTER filter
)

DRV_AK4384_MuteOff Function

Disables AK4384 output for soft mute.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_MuteOff(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables AK4384 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

    DRV_AK4384_MuteOff(myAK4384Handle); //AK4384 output soft mute disabled
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_MuteOff(DRV_HANDLE handle)

DRV_AK4384_MuteOn Function

Allows AK4384 output for soft mute on.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_MuteOn(DRV_HANDLE handle);
```

Returns

None.

Description

This function Enables AK4384 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_MuteOn(myAK4384Handle);  //AK4384 output soft muted
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_MuteOn(DRV_HANDLE handle);

DRV_AK4384_SamplingRateGet Function

This function gets the sampling rate set on the DAC AK4384.

Implementation: Dynamic

File

drv_ak4384.h

C

```
uint32_t DRV_AK4384_SamplingRateGet(DRV_HANDLE handle);
```

Returns

None.

Description

This function gets the sampling rate set on the DAC AK4384.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
uint32_t baudRate;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

baudRate = DRV_AK4384_SamplingRateGet(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_AK4384_SamplingRateGet(DRV_HANDLE handle)

DRV_AK4384_SamplingRateSet Function

This function sets the sampling rate of the media stream.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

Returns

None.

Description

This function sets the media sampling rate for the client handle.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_SamplingRateSet(myAK4384Handle, 48000);  //Sets 48000 media sampling rate
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
baudRate	Baud Rate to be set

Function

void DRV_AK4384_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

DRV_AK4384_SlowRollOffFilterDisable Function

Disables Slow Roll-off filter function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_SlowRollOffFilterDisable(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables Slow Roll-off filter function. Sharp Roll-off filter function gets enabled.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_SlowRollOffFilterDisable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_SlowRollOffFilterDisable(DRV_HANDLE handle);

DRV_AK4384_SlowRollOffFilterEnable Function

Enables Slow Roll-off filter function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_SlowRollOffFilterEnable(DRV_HANDLE handle);
```

Returns

None.

Description

This function enables Slow Roll-off filter function.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_SlowRollOffFilterEnable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_SlowRollOffFilterEnable(DRV_HANDLE handle);

DRV_AK4384_VolumeGet Function

This function gets the volume for AK4384 Codec.

Implementation: Dynamic

File

drv_ak4384.h

C

```
uint8_t DRV_AK4384_VolumeGet(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan);
```

Returns

None.

Description

This functions gets the current volume programmed to the DAC AK4384.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

  volume = DRV_AK4384_VolumeGet(myAK4384Handle, DRV_AK4384_CHANNEL_LEFT_RIGHT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to get.

Function

uint8_t DRV_AK4384_VolumeGet(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan)

DRV_AK4384_VolumeSet Function

This function sets the volume for AK4384 Codec.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_VolumeSet(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan, uint8_t volume);
```

Returns

None.

Description

This functions sets the volume value from 0-255, which can attenuate from 0 dB to –48 dB and mute.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

    DRV_AK4384_VolumeSet(myAK4384Handle, DRV_AK4384_CHANNEL_LEFT_RIGHT, 120); //Step 120 volume
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set
volume	volume value from 0-255, which can attenuate from 0 dB to –48 dB and mute

Function

void DRV_AK4384_VolumeSet(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan, uint8_t volume)

DRV_AK4384_ZeroDetectDisable Function

Disables AK4384 channel-independent zeros detect function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ZeroDetectDisable(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables AK4384 channel-independent zeros detect function.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ZeroDetectDisable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_ZeroDetectDisable(DRV_HANDLE handle)

DRV_AK4384_ZeroDetectEnable Function

Enables AK4384 channel-independent zeros detect function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ZeroDetectEnable(DRV_HANDLE handle);
```

Returns

None.

Description

This function enables AK4384 channel-independent zeros detect function.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ZeroDetectEnable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_ZeroDetectEnable(DRV_HANDLE handle)

DRV_AK4384_ZeroDetectInvertDisable Function

Disables inversion of polarity for zero detect function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ZeroDetectInvertDisable(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables inversion of polarity for zero detect function. DZF goes “H” at Zero Detection.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ZeroDetectInvertDisable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_ZeroDetectInvertDisable(DRV_HANDLE handle)

DRV_AK4384_ZeroDetectInvertEnable Function

Enables inversion of polarity for zero detect function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ZeroDetectInvertEnable(DRV_HANDLE handle);
```

Returns

None.

Description

This function enables inversion of polarity for zero detect function. DZF goes “L” at Zero Detection

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ZeroDetectInvertEnable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_ZeroDetectInvertEnable(DRV_HANDLE handle)

DRV_AK4384_ZeroDetectModeSet Function

Sets mode of AK4384 channel-independent zeros detect function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ZeroDetectModeSet(DRV_HANDLE handle, DRV_AK4384_ZERO_DETECT_MODE zdMode);
```

Returns

None.

Description

This function sets mode of AK4384 channel-independent zeros detect function

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ZeroDetectModeSet(myAK4384Handle, DRV_AK4384_ZERO_DETECT_MODE_ANDED);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
zdMode	Specifies zero detect function mode.

Function

```
void DRV_AK4384_ZeroDetectModeSet
(
    DRV_HANDLE handle,
    DRV_AK4384_ZERO_DETECT_MODE zdMode
)
```

d) Data Transfer Functions

DRV_AK4384_BufferAddWrite Function

Schedule a non-blocking driver write operation.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_BufferAddWrite(const DRV_HANDLE handle, DRV_AK4384_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4384_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4384_BUFFER_HANDLE_INVALID if:

- a buffer could not be allocated to the request
- the input buffer pointer is NULL
- the buffer size is '0'
- the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4384_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4384_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4384 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4384 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 device instance and the DRV_AK4384_Status must have returned SYS_STATUS_READY.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK4384_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4384_BUFFER_HANDLE bufferHandle;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

// Client registers an event handler with driver

DRV_AK4384_BufferEventHandlerSet(myAK4384Handle,
```

```
                APP_AK4384BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4384_BufferAddWrite(myAK4384handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4384_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4384BufferEventHandler(DRV_AK4384_BUFFER_EVENT event,
        DRV_AK4384_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4384_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4384_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4384 instance as return by the DRV_AK4384_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4384_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_AK4384_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

DRV_AK4384_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK4384_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_AK4384_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4384_BUFFER_HANDLE bufferHandle;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

// Client registers an event handler with driver

DRV_AK4384_BufferEventHandlerSet(myAK4384Handle,
                APP_AK4384BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4384_BufferAddWrite(myAK4384handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4384_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4384BufferEventHandler(DRV_AK4384_BUFFER_EVENT event,
        DRV_AK4384_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4384_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
```

```
                break;

            case DRV_AK4384_BUFFER_EVENT_ERROR:

                // Error handling here.
                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4384_BufferEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4384_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4384_BufferCombinedQueueSizeGet Function

This function returns the number of bytes queued (to be processed) in the buffer queue.

Implementation: Dynamic

File

drv_ak4384.h

C

```
size_t DRV_AK4384_BufferCombinedQueueSizeGet(DRV_HANDLE handle);
```

Returns

Returns the number of the bytes that have been processed for this buffer. Returns 0 for an invalid or an expired client handle.

Description

This function returns the number of bytes queued (to be processed) in the buffer queue associated with the driver instance to which the calling client belongs. The client can use this function to know number of bytes that is in the queue to be transmitted.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

One of DRV_AK4384_BufferAddRead/DRV_AK4384_BufferAddWrite function must have been called and buffers should have been queued for transmission.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
size_t bufferQueuedSize;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4384_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_AK4384_Open function.

// Client registers an event handler with driver. This is done once

DRV_AK4384_BufferEventHandlerSet(myAK4384Handle, APP_AK4384BufferEventHandle,
                                                (uintptr_t)&myAppObj);

DRV_AK4384_BufferAddRead(myAK4384handle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4384_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// The data is being processed after adding the buffer to the queue.
// The user can get to know dynamically available data in the queue to be
// transmitted by calling DRV_AK4384_BufferCombinedQueueSizeGet
bufferQueuedSize = DRV_AK4384_BufferCombinedQueueSizeGet(myAK4384Handle);
```

Parameters

Parameters	Description
handle	Opened client handle associated with a driver object.

Function

size_t DRV_AK4384_BufferCombinedQueueSizeGet(DRV_HANDLE handle)

DRV_AK4384_BufferProcessedSizeGet Function

This function returns number of bytes that have been processed for the specified buffer.

Implementation: Dynamic

File

drv_ak4384.h

C

```
size_t DRV_AK4384_BufferProcessedSizeGet(DRV_AK4384_BUFFER_HANDLE bufferHandle);
```

Returns

Returns the number of the bytes that have been processed for this buffer. Returns 0 for an invalid or an expired buffer handle.

Description

This function returns number of bytes that have been processed for the specified buffer. The client can use this function, in a case where the buffer has terminated due to an error, to obtain the number of bytes that have been processed. If this function is called on a invalid buffer handle, or if the buffer handle has expired, the function returns 0.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified I2S driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

One of DRV_AK4384_BufferAddRead, DRV_AK4384_BufferAddWrite function must have been called and a valid buffer handle returned.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4384_BUFFER_HANDLE bufferHandle;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

// Client registers an event handler with driver. This is done once

DRV_AK4384_BufferEventHandlerSet(myAK4384Handle, APP_AK4384BufferEventHandle,
                                                        (uintptr_t)&myAppObj);

DRV_AK4384_BufferAddRead(myAK4384handle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4384_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_AK4384BufferEventHandler(DRV_AK4384_BUFFER_EVENT event,
        DRV_AK4384_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_AK4384_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4384_BUFFER_EVENT_ERROR:

            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred.

            processedBytes = DRV_AK4384_BufferProcessedSizeGet(bufferHandle);

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
bufferhandle	Handle of the buffer of which the processed number of bytes to be obtained.

Function

size_t DRV_AK4384_BufferProcessedSizeGet(DRV_AK4384_BUFFER_HANDLE bufferHandle)

e) Other Functions

DRV_AK4384_CommandEventHandlerSet Function

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_CommandEventHandlerSet(DRV_HANDLE handle, const
DRV_AK4384_COMMAND_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK4384_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "AK4384 CODEC Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

// Client registers an event handler with driver

DRV_AK4384_CommandEventHandlerSet(myAK4384Handle,
                APP_AK4384CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK4384_DeEmphasisFilterSet(myAK4384Handle, DRV_AK4384_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK4384CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
```

```
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4384_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4384_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4384_VersionGet Function

Returns the version of the AK4384 driver.

Implementation: Dynamic

File

drv_ak4384.h

C

```
uint32_t DRV_AK4384_VersionGet();
```

Returns

Returns the version of AK4384 driver.

Description

The version number returned from the DRV_AK4384_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Remarks

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
uint32_t ak4384version;
ak4384version = DRV_AK4384_VersionGet();
```

Function

uint32_t DRV_AK4384_VersionGet(void)

DRV_AK4384_VersionStrGet Function

Returns the version of AK4384 driver in string format.

Implementation: Dynamic

File

drv_ak4384.h

C

```
int8_t* DRV_AK4384_VersionStrGet();
```

Returns

returns a string containing the version of AK4384 driver.

Description

The DRV_AK4384_VersionStrGet function returns a string in the format: ".[.][]" Where: is the AK4384 driver's version number. is the AK4384 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals '00'). is an optional release type ('a' for alpha, 'b' for beta not the entire word spelled out) that is not included if the release is a production version (i.e., not an alpha or beta).

The String does not contain any spaces.

Remarks

None.

Preconditions

None.

Example 1

"0.03a" "1.00"

Example 2

```
int8_t *ak4384string;
ak4384string = DRV_AK4384_VersionStrGet();
```

Function

int8_t* DRV_AK4384_VersionStrGet(void)

f) Data Types and Constants

DRV_AK4384_AUDIO_DATA_FORMAT Enumeration

Identifies the Serial Audio data interface format.

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_AUDIO_DATA_FORMAT_16BIT_RIGHT_JUSTIFIED = 0,
  DRV_AK4384_AUDIO_DATA_FORMAT_20BIT_RIGHT_JUSTIFIED,
  DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_LEFT_JUSTIFIED,
  DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_I2S,
  DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_RIGHT_JUSTIFIED
} DRV_AK4384_AUDIO_DATA_FORMAT;
```

Members

Members	Description
DRV_AK4384_AUDIO_DATA_FORMAT_16BIT_RIGHT_JUSTIFIED = 0	16 bit Right Justified Audio data format
DRV_AK4384_AUDIO_DATA_FORMAT_20BIT_RIGHT_JUSTIFIED	20 bit Right Justified Audio data format
DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_LEFT_JUSTIFIED	24 bit Left Justified Audio data format
DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_I2S	24 bit I2S Audio data format
DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_RIGHT_JUSTIFIED	24 bit Right Justified Audio data format

Description

AK4384 Audio data format

This enumeration identifies Serial Audio data interface format.

Remarks

None.

DRV_AK4384_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_BUFFER_EVENT_COMPLETE,
  DRV_AK4384_BUFFER_EVENT_ERROR,
  DRV_AK4384_BUFFER_EVENT_ABORT
} DRV_AK4384_BUFFER_EVENT;
```

Members

Members	Description
DRV_AK4384_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_AK4384_BUFFER_EVENT_ERROR	Error while processing the request

DRV_AK4384_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

AK4384 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_AK4384_BufferAddWrite function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_AK4384_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_AK4384_BUFFER_EVENT_HANDLER Type

Pointer to a AK4384 Driver Buffer Event handler function.

File

drv_ak4384.h

C

```
typedef void (* DRV_AK4384_BUFFER_EVENT_HANDLER)(DRV_AK4384_BUFFER_EVENT event,
DRV_AK4384_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

AK4384 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK4384 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_AK4384_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK4384_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_AK4384_BufferProcessedSizeGet function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4384_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver (I2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_AK4384_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_AK4384_BUFFER_EVENT event,
                               DRV_AK4384_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK4384_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK4384_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4384_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

File

drv_ak4384.h

C

```
typedef uintptr_t DRV_AK4384_BUFFER_HANDLE;
```

Description

AK4384 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK4384_BufferAddWrite function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_AK4384_CHANNEL Enumeration

Identifies Left/Right Audio channel

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_CHANNEL_LEFT,
  DRV_AK4384_CHANNEL_RIGHT,
  DRV_AK4384_CHANNEL_LEFT_RIGHT,
  DRV_AK4384_NUMBER_OF_CHANNELS
} DRV_AK4384_CHANNEL;
```

Description

AK4384 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

DRV_AK4384_COMMAND_EVENT_HANDLER Type

Pointer to a AK4384 Driver Command Event Handler Function

File

drv_ak4384.h

C

```
typedef void (* DRV_AK4384_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

Returns

None.

Description

AK4384 Driver Command Event Handler Function

This data type defines the required function signature for the AK4384 driver command event handling callback function.

A command is a control instruction to the AK4384 Codec. For example, Mute ON/OFF, Zero Detect Enable/Disable, etc.

A client must register a pointer to a command event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4384_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_AK4384CommandEventHandler( uintptr_t context )
```

```
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4384_DEEMPHASIS_FILTER Enumeration

Identifies de-emphasis filter function.

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_DEEMPHASIS_FILTER_44_1KHZ,
  DRV_AK4384_DEEMPHASIS_FILTER_OFF,
  DRV_AK4384_DEEMPHASIS_FILTER_48KHZ,
  DRV_AK4384_DEEMPHASIS_FILTER_32KHZ
} DRV_AK4384_DEEMPHASIS_FILTER;
```

Members

Members	Description
DRV_AK4384_DEEMPHASIS_FILTER_44_1KHZ	De-Emphasis filter for 44.1kHz.
DRV_AK4384_DEEMPHASIS_FILTER_OFF	De-Emphasis filter Off This is the default setting.
DRV_AK4384_DEEMPHASIS_FILTER_48KHZ	De-Emphasis filter for 48kHz.
DRV_AK4384_DEEMPHASIS_FILTER_32KHZ	De-Emphasis filter for 32kHz.

Description

AK4384 De-Emphasis Filter

This enumeration identifies the settings for de-emphasis filter function.

Remarks

None.

DRV_AK4384_INIT Structure

Defines the data required to initialize or reinitialize the AK4384 driver.

File

drv_ak4384.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX spiDriverModuleIndex;
  SYS_MODULE_INDEX i2sDriverModuleIndex;
  uint8_t volume;
  DRV_AK4384_MCLK_MODE mclkMode;
```

```
} DRV_AK4384_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX spiDriverModuleIndex;	Identifies control module(SPI) driver ID for control interface of Codec
SYS_MODULE_INDEX i2sDriverModuleIndex;	Identifies data module(I2S) driver ID for data interface of Codec
uint8_t volume;	Volume
DRV_AK4384_MCLK_MODE mclkMode;	Set MCLK mode.

Description

AK4384 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK4384 Codec driver.

Remarks

None.

DRV_AK4384_MCLK_MODE Enumeration

Identifies the mode of master clock to AK4384 DAC.

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_MCLK_MODE_MANUAL,
  DRV_AK4384_MCLK_MODE_AUTO
} DRV_AK4384_MCLK_MODE;
```

Members

Members	Description
DRV_AK4384_MCLK_MODE_MANUAL	Master clock frequency mode Manual
DRV_AK4384_MCLK_MODE_AUTO	Master clock frequency mode Auto This is the default mode.

Description

AK4384 Master clock frequency mode

This enumeration identifies mode of master clock to AK4384 DAC. In Manual Setting Mode, the sampling speed is set by setting DFS0/1 bits in Control Register 2. The frequency of MCLK at each sampling speed is set automatically. In Auto Setting Mode, the MCLK frequency is detected automatically

Remarks

None.

DRV_AK4384_ZERO_DETECT_MODE Enumeration

Identifies Zero Detect Function mode

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_ZERO_DETECT_MODE_CHANNEL_SEPARATED,
  DRV_AK4384_ZERO_DETECT_MODE_ANDED
} DRV_AK4384_ZERO_DETECT_MODE;
```

Members

Members	Description
DRV_AK4384_ZERO_DETECT_MODE_CHANNEL_SEPARATED	Zero Detect channel separated. When the input data at each channel is continuously zeros for 8192 LRCK cycles, DZF pin of each channel goes to “H” This is the default mode.
DRV_AK4384_ZERO_DETECT_MODE_ANDED	Zero Detect Anded DZF pins of both channels go to “H” only when the input data at both channels are continuously zeros for 8192 LRCK cycles

Description

AK4384 Zero Detect mode

This enumeration identifies the mode of zero detect function

Remarks

None.

DRV_AK4384_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_ak4384.h

C

```
#define DRV_AK4384_BUFFER_HANDLE_INVALID ((DRV_AK4384_BUFFER_HANDLE)(-1))
```

Description

AK4384 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_AK4384_BufferAddWrite function if the buffer add request was not successful.

Remarks

None.

DRV_AK4384_COUNT Macro

Number of valid AK4384 driver indices.

File

drv_ak4384.h

C

```
#define DRV_AK4384_COUNT
```

Description

AK4384 Driver Module Count

This constant identifies the maximum number of AK4384 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of AK4384 instances on this microcontroller.

Remarks

This value is device-specific.

DRV_AK4384_INDEX_0 Macro

AK4384 driver index definitions.

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_0 0
```

Description

Driver AK4384 Module Index

These constants provide AK4384 driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_AK4384_Initialize and DRV_AK4384_Open routines to identify the driver instance in use.

DRV_AK4384_INDEX_1 Macro

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_1 1
```

Description

This is macro DRV_AK4384_INDEX_1.

DRV_AK4384_INDEX_2 Macro

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_2 2
```

Description

This is macro DRV_AK4384_INDEX_2.

DRV_AK4384_INDEX_3 Macro

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_3 3
```

Description

This is macro DRV_AK4384_INDEX_3.

DRV_AK4384_INDEX_4 Macro

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_4 4
```

Description

This is macro DRV_AK4384_INDEX_4.

DRV_AK4384_INDEX_5 Macro

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_5 5
```

Description

This is macro DRV_AK4384_INDEX_5.

Files

Files

Name	Description
drv_ak4384.h	AK4384 Codec Driver Interface header file
drv_ak4384_config_template.h	AK4384 Codec Driver Configuration Template.

Description

This section lists the source and header files used by the AK4384Codec Driver Library.

drv_ak4384.h

AK4384 Codec Driver Interface header file

Enumerations

	Name	Description
	DRV_AK4384_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4384_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4384_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4384_DEEMPHASIS_FILTER	Identifies de-emphasis filter function.
	DRV_AK4384_MCLK_MODE	Identifies the mode of master clock to AK4384 DAC.
	DRV_AK4384_ZERO_DETECT_MODE	Identifies Zero Detect Function mode

Functions

	Name	Description
	DRV_AK4384_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_AK4384_BufferCombinedQueueSizeGet	This function returns the number of bytes queued (to be processed) in the buffer queue. **Implementation:** Dynamic
	DRV_AK4384_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_AK4384_BufferProcessedSizeGet	This function returns number of bytes that have been processed for the specified buffer. **Implementation:** Dynamic
	DRV_AK4384_ChannelOutputInvertDisable	Disables output polarity of the selected Channel. **Implementation:** Dynamic
	DRV_AK4384_ChannelOutputInvertEnable	Enables output polarity of the selected channel. **Implementation:** Dynamic
	DRV_AK4384_Close	Closes an opened-instance of the AK4384 driver. **Implementation:** Dynamic
	DRV_AK4384_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. **Implementation:** Dynamic
	DRV_AK4384_DeEmphasisFilterSet	Allows specifies enabling of digital de-emphasis filter. **Implementation:** Dynamic
	DRV_AK4384_Deinitialize	Deinitializes the specified instance of the AK4384 driver module. **Implementation:** Dynamic
	DRV_AK4384_Initialize	Initializes hardware and data for the instance of the AK4384 DAC module. **Implementation:** Dynamic
	DRV_AK4384_MuteOff	Disables AK4384 output for soft mute. **Implementation:** Dynamic
	DRV_AK4384_MuteOn	Allows AK4384 output for soft mute on. **Implementation:** Dynamic

	Name	Description
	DRV_AK4384_Open	Opens the specified AK4384 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_AK4384_SamplingRateGet	This function gets the sampling rate set on the DAC AK4384. **Implementation:** Dynamic
	DRV_AK4384_SamplingRateSet	This function sets the sampling rate of the media stream. **Implementation:** Dynamic
	DRV_AK4384_SlowRollOffFilterDisable	Disables Slow Roll-off filter function. **Implementation:** Dynamic
	DRV_AK4384_SlowRollOffFilterEnable	Enables Slow Roll-off filter function. **Implementation:** Dynamic
	DRV_AK4384_Status	Gets the current status of the AK4384 driver module. **Implementation:** Dynamic
	DRV_AK4384_Tasks	Maintains the driver's control and data interface state machine. **Implementation:** Dynamic
	DRV_AK4384_VersionGet	Returns the version of the AK4384 driver. **Implementation:** Dynamic
	DRV_AK4384_VersionStrGet	Returns the version of AK4384 driver in string format. **Implementation:** Dynamic
	DRV_AK4384_VolumeGet	This function gets the volume for AK4384 Codec. **Implementation:** Dynamic
	DRV_AK4384_VolumeSet	This function sets the volume for AK4384 Codec. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectDisable	Disables AK4384 channel-independent zeros detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectEnable	Enables AK4384 channel-independent zeros detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectInvertDisable	Disables inversion of polarity for zero detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectInvertEnable	Enables inversion of polarity for zero detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectModeSet	Sets mode of AK4384 channel-independent zeros detect function. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_AK4384_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4384_COUNT	Number of valid AK4384 driver indices.
	DRV_AK4384_INDEX_0	AK4384 driver index definitions.
	DRV_AK4384_INDEX_1	This is macro DRV_AK4384_INDEX_1.
	DRV_AK4384_INDEX_2	This is macro DRV_AK4384_INDEX_2.
	DRV_AK4384_INDEX_3	This is macro DRV_AK4384_INDEX_3.
	DRV_AK4384_INDEX_4	This is macro DRV_AK4384_INDEX_4.
	DRV_AK4384_INDEX_5	This is macro DRV_AK4384_INDEX_5.

Structures

	Name	Description
	DRV_AK4384_INIT	Defines the data required to initialize or reinitialize the AK4384 driver.

Types

	Name	Description
	DRV_AK4384_BUFFER_EVENT_HANDLER	Pointer to a AK4384 Driver Buffer Event handler function.
	DRV_AK4384_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4384_COMMAND_EVENT_HANDLER	Pointer to a AK4384 Driver Command Event Handler Function

Description

AK4384 Codec Driver Interface

The AK4384 Codec device driver interface provides a simple interface to manage the AK4384 106 dB 192 kHz 24-Bit DAC that can be interfaced Microchip Microcontroller. This file provides the interface definition for the AK4384 Codec device driver.

File Name

drv_ak4384.h

Company

Microchip Technology Inc.

drv_ak4384_config_template.h

AK4384 Codec Driver Configuration Template.

Macros

	Name	Description
	DRV_AK4384_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1, and 48K sampling frequency
	DRV_AK4384_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4384_CONTROL_CLOCK	Sets up clock frequency for the control interface (SPI)
	DRV_AK4384_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4384_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4384_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1 and 48K sampling frequency
	DRV_AK4384_TIMER_DRIVER_MODULE_INDEX	Identifies the Timer Module Index for custom virtual SPI driver implementation.
	DRV_AK4384_TIMER_PERIOD	Identifies the period for the bit bang timer.

Description

AK4384 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_ak4384_config_template.h

Company

Microchip Technology Inc.

AK4642 Codec Driver Library

This topic describes the AK4642 Codec Driver Library.

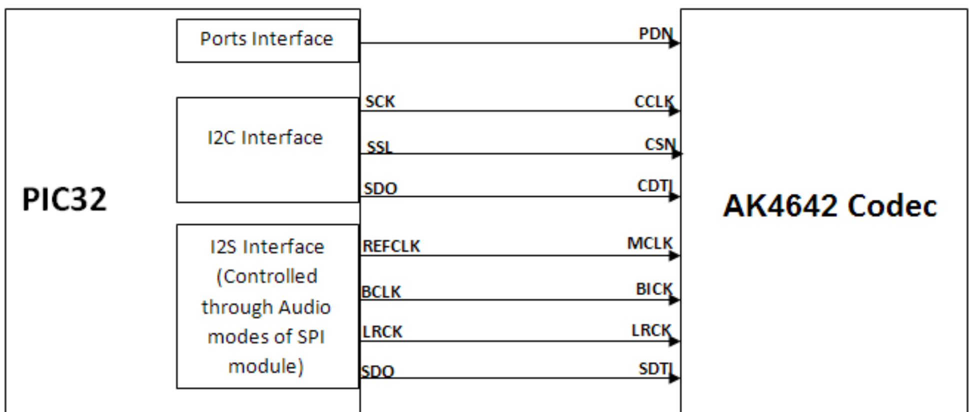
Introduction

This library provides an interface to manage the AK4642 Codec that is serially interfaced to a Microchip microcontroller for providing Audio Solutions.

Description

The AK4642 module is 16/24-bit Audio Codec from Asahi Kasei Microdevices Corporation. The AK4642 can be interfaced to Microchip microcontrollers through I2C and I2S serial interfaces. The I2C interface is used for control command transfer. The I2S interface is used for Audio data output.

A typical interface of AK4642 to a Microchip PIC32 device is provided in the following diagram:



Features

The AK4642 Codec Driver supports the following features:

- Audio Interface Format: MSB first
- ADC: 16-bit MSB justified, 16/24-bit I2S
- DAC: 16-bit MSB justified, 16bit LSB justified, 16/24-bit I2S
- Sampling Frequency Range: 8 kHz to 48 kHz
- Digital Volume Control: +12dB ~ .115dB, 0.5dB Step
- SoftMute: On and Off
- Master Clock Frequencies: 32 fs/64 fs/128fs/256fs

Using the Library

This topic describes the basic architecture of the AK4642 Codec Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_ak4642.h`

The interface to the AK4642 Codec Driver library is defined in the `drv_ak4642.h` header file. Any C language source (`.c`) file that uses the AK4642 Codec Driver library should include this header.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

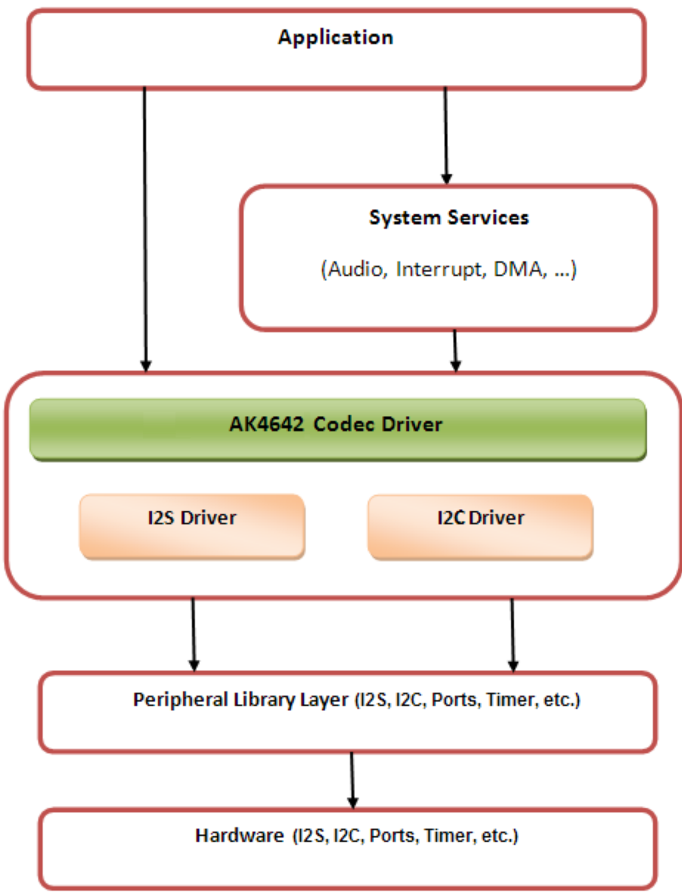
Abstraction Model

This library provides a low-level abstraction of the AK4642 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the AK4642 Codec Driver is positioned in the MPLAB Harmony framework. The AK4642 Codec Driver uses the SPI and I2S drivers for control and audio data transfers to the AK4642 module.

AK4642 Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The AK4642 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK4642 DAC module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK4642 Codec Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open and close functions.
Codec Specific Functions	Provides functions that are codec specific.
Data Transfer Functions	Provides data transfer functions.
Other Functions	Provides driver specific miscellaneous functions such as sampling rate setting, control command functions, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the AK4642 Codec Driver Library.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

System Access

This topic provides information on system initialization, implementations, and provides a system access code example.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the AK4642 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK4642_INIT or by using Initialization Overrides) that are supported by the specific AK4642 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver.
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver.
- Sampling rate
- Master clock detection mode
- Power down pin port initialization

The DRV_AK4642_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ AK4642_Deinitialize, DRV_ AK4642_Status and DRV_I2S_Tasks.

Implementations

The AK4642 Codec Driver can have the following implementations:

Implementation	Description	MPLAB Harmony Components
Implementation 1	Dedicated hardware for control (I2C) and data (I2S) interface.	Standard MPLAB Harmony drivers for I2C and I2S interfaces.
Implementation 2	Dedicated hardware for data (I2S) interface. Ports pins for control interface.	Standard MPLAB Harmony drivers for I2S interface. Virtual MPLAB Harmony drivers for I2C interface.

Example:

```
DRV_AK4642_INIT drvak4642Init =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .i2sDriverModuleIndex = DRV_AK4642_I2S_DRIVER_MODULE_INDEX_IDX0,
    .i2cDriverModuleIndex = DRV_AK4642_I2C_DRIVER_MODULE_INDEX_IDX0,
    .volume = DRV_AK4642_VOLUME,
};

/*
  The I2C and I2S module index should be same as the one used in
  initializing the I2C and I2S drivers.
 */

ak4642DevObject = DRV_AK4642_Initialize(DRV_AK4642_INDEX_0, (SYS_MODULE_INIT *) &drvak4642Init);
if (SYS_MODULE_OBJ_INVALID == ak4642DevObject)
{
```

```
    // Handle error
}
```

Task Routine

The DRV_AK4642_Tasks will be called from the System Task Service.

Client Access

This topic describes client access and includes a code example.

Description

For the application to start using an instance of the module, it must call the DRV_AK4642_Open function. The DRV_AK4642_Open provides a driver handle to the AK4642 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK4642_Deinitialize, the application must call the DRV_AK4642_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Note: It is necessary to check the status of driver initialization before opening a driver instance. The status of the AK4642 Codec Driver can be known by calling DRV_AK4642_Status.

Example:

```
DRV_HANDLE handle;
SYS_STATUS ak4642Status;
    ak4642Status = DRV_AK4642_Status(sysObjects.ak4642DevObject);
        if (SYS_STATUS_READY == ak4642Status)
        {
                // The driver can now be opened.
                appData.ak4642Client.handle = DRV_AK4642_Open
                                             (DRV_AK4642_INDEX_0,
                                              DRV_IO_INTENT_WRITE |
                                              DRV_IO_INTENT_EXCLUSIVE );
                if(appData.ak4642Client.handle != DRV_HANDLE_INVALID)
                {
                    appData.state = APP_STATE_AK4642_SET_BUFFER_HANDLER;
                }
                else
                {
                    SYS_DEBUG(0, "Find out what's wrong \r\n");
                }
    }
    else
    {
        /* AK4642 Driver Is not ready */
                ;
    }
```

Client Operations

This topic describes client operations and provides a code example.

Description

Client operations provide the API interface for control command and audio data transfer to the AK4642 Codec.

The following AK4642 Codec specific control command functions are provided:

- DRV_AK4642_SamplingRateSet
- DRV_AK4642_SamplingRateGet
- DRV_AK4642_VolumeSet

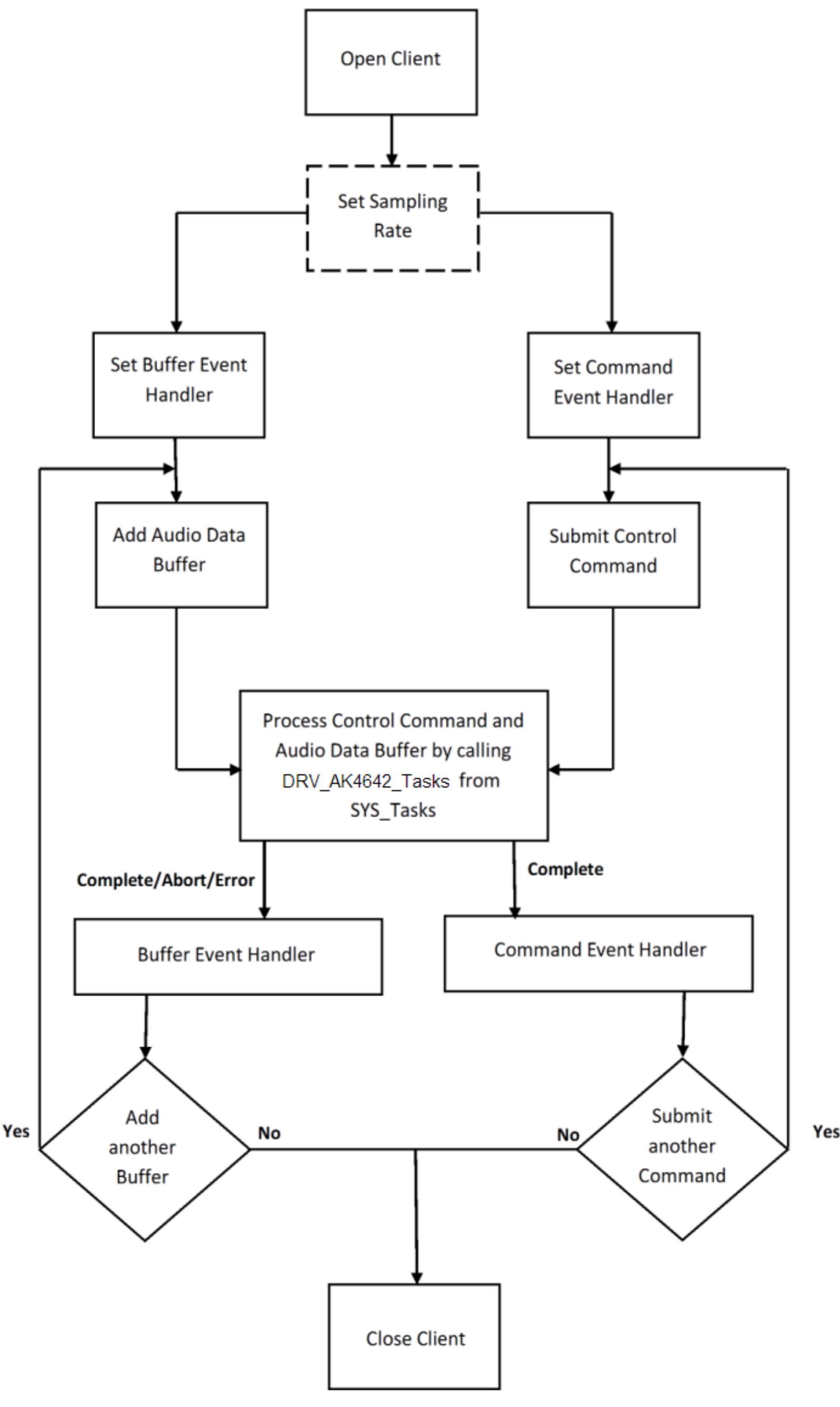
- DRV_AK4642_VolumeGet
- DRV_AK4642_MuteOn
- DRV_AK4642_MuteOff

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the AK4642 Codec. A notification for the submitted requests can be received by registering a command callback event with the driver. The driver notifies by calling the callback on successfully transmitting the command to the AK4642 Codec module.

DRV_AK4642_BufferAddWrite, DRV_AK4642_BufferAddRead, and DRV_AK4642_BufferAddWriteRead are buffered data operation functions. These functions schedule non-blocking audio data transfer operations. The function adds the request to the hardware instance queues and returns a buffer handle. The requesting client also registers a callback event with the driver. The driver notifies the client with DRV_AK4642_BUFFER_EVENT_COMPLETE, DRV_AK4642_BUFFER_EVENT_ERROR, or DRV_AK4642_BUFFER_EVENT_ABORT events.

The submitted control commands and audio buffer add requests are processed under DRV_AK4642_Tasks function. This function is called from the SYS_Tasks routine.

The following diagram illustrates the control commands and audio buffered data operations.



Note: It is not necessary to close and reopen the client between multiple transfers.

An application using the buffered functionality needs to perform the following steps:

1. The system should have completed necessary setup and initializations.

2. The I2S driver object should have been initialized by calling DRV_I2S_Initialize.
3. The I2C driver object should have been initialized by calling DRV_I2C_Initialize.
4. The AK4642 driver object should be initialized by calling DRV_AK4642_Initialize.
5. The necessary sampling rate value should be set up by calling DRV_AK4642_ SamplingRateSet.
6. Register buffer event handler for the client handle by calling DRV_AK4642_BufferEventHandlerSet.
7. Register command event handler for the client handle by calling DRV_AK4642_CommandEventHandlerSet.
8. Submit a command by calling specific command API.
9. Add a buffer to initiate the data transfer by calling DRV_AK4642_BufferAddWrite, DRV_AK4642_BufferAddRead, and DRV_AK4642_BufferAddWriteRead.
10. The submitted command and Audio data processing happens b calling DRV_AK4642_Tasks from SYS_Tasks.
11. Repeat steps 9 through 10 to handle multiple buffer transmission and reception.
12. When the client is done, it can use DRV_AK4642_Close to close the client handle.

Example:

```
typedef enum
{
    APP_STATE_AK4642_OPEN,
    APP_STATE_AK4642_SET_BUFFER_HANDLER,
    APP_STATE_AK4642_ADD_FIRST_BUFFER_READ,
    APP_STATE_AK4642_ADD_BUFFER_OUT,
    APP_STATE_AK4642_ADD_BUFFER_IN,
    APP_STATE_AK4642_WAIT_FOR_BUFFER_COMPLETE,
} APP_STATES;

typedef struct
{
    DRV_HANDLE handle;
    DRV_AK4642_BUFFER_HANDLE writereadBufHandle;
    DRV_AK4642_BUFFER_EVENT_HANDLER bufferEventHandler;
    uintptr_t context;
    uint8_t *txbufferObject;
    uint8_t *rxbufferObject;
    size_t bufferSize;

} APP_AK4642_CLIENT;

typedef struct
{
    /* Application's current state*/
    APP_STATES state;
    /* USART client handle */
    APP_AK4642_CLIENT ak4642Client;
} APP_DATA;
APP_DATA appData;
SYS_MODULE_OBJ ak4642DevObject;
DRV_AK4642_INIT drvak4642Init =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .i2sDriverModuleIndex = DRV_AK4642_I2S_DRIVER_MODULE_INDEX_IDX0,
    .i2cDriverModuleIndex = DRV_AK4642_I2C_DRIVER_MODULE_INDEX_IDX0,
    .volume = DRV_AK4642_VOLUME,
};

void SYS_Initialize(void * data)
{
    /* Initialize Drivers */
    DRV_I2C0_Initialize();
    sysObj.drvI2S0 = DRV_I2S_Initialize(DRV_I2S_INDEX_0, (SYS_MODULE_INIT *)
                                        &drvI2S0InitData);

    sysObj.drvak4642Codec0 = DRV_AK4642_Initialize(DRV_AK4642_INDEX_0,
                             (SYS_MODULE_INIT *)&drvak4642Codec0InitData);

    /* Initialize System Services */
    SYS_INT_Initialize();
```

```
}

void APP_Tasks (void )
{
    switch(appData.state)
    {
        case APP_STATE_AK4642_OPEN:
        {
            /* A client opens the driver object to get an Handle */
            appData.ak4642Client.handle = DRV_AK4642_Open(DRV_AK4642_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
            if(appData.ak4642Client.handle != DRV_HANDLE_INVALID)
            {
                appData.state = APP_STATE_AK4642_SET_BUFFER_HANDLER;
            }
            else
            {
                /* Got an Invalid Handle.  Wait for AK4642 to Initialize */
                ;
            }
        }
        break;

        /* Set a handler for the audio buffer completion event */
        case APP_STATE_AK4642_SET_BUFFER_HANDLER:
        {

           DRV_AK4642_BufferEventHandlerSet(appData.ak4642Client.handle,
                    appData.ak4642Client.bufferEventHandler,
                    appData.ak4642Client.context);

            appData.state = APP_STATE_AK4642_ADD_FIRST_BUFFER_READ;
        }
        break;

        case APP_STATE_AK4642_ADD_FIRST_BUFFER_READ:
        {
            DRV_AK4642_BufferAddWriteRead(appData.ak4642Client.handle,
                    &appData.ak4642Client.writeReadBufHandle,
                    appData.ak4642Client.txbufferObject,
                    appData.ak4642Client.rxbufferObject,
                    appData.ak4642Client.bufferSize);
            if(appData.ak4642Client.writeReadBufHandle != DRV_AK4642_BUFFER_HANDLE_INVALID)
            {
                appData.state = APP_STATE_AK4642_WAIT_FOR_BUFFER_COMPLETE;
            }
            else
            {
                SYS_DEBUG(0, "Find out what is wrong \r\n");
            }

        }
        break;
        /* Add an audio buffer to the ak4642 driver to be transmitted to
         * AK4642 CODEC */
        case APP_STATE_AK4642_ADD_BUFFER_OUT:
        {

            DRV_AK4642_BufferAddWrite(appData.ak4642Client.handle, &appData.ak4642Client.writeBufHandle,
            appData.ak4642Client.txbufferObject, appData.ak4642Client.bufferSize);
            if(appData.ak4642Client.writeBufHandle != DRV_AK4642_BUFFER_HANDLE_INVALID)
            {

                appData.state = APP_STATE_AK4642_WAIT_FOR_BUFFER_COMPLETE;
            }
            else
            {
                SYS_DEBUG(0, "Find out what is wrong \r\n");
            }
        }
```

```
        break;
        /* Add an audio buffer to the ak4642 driver to be received
         * AK4642 CODEC */
        case APP_STATE_AK4642_ADD_BUFFER_IN:
        {

            DRV_AK4642_BufferAddRead(appData.ak4642Client.handle, &appData.ak4642Client.readBufHandle,
            appData.ak4642Client.rxbufferObject, appData.ak4642Client.bufferSize);


            if(appData.ak4642Client.readBufHandle != DRV_AK4642_BUFFER_HANDLE_INVALID)
            {
                appData.state = APP_STATE_AK4642_ADD_BUFFER_OUT;
            }
            else
            {
                SYS_DEBUG(0, "Find out what is wrong \r\n");
            }
        }
        break;
        /* Audio data Transmission under process */
        case APP_STATE_AK4642_WAIT_FOR_BUFFER_COMPLETE:
        {
            /*Do nothing*/
        }
        break;

        default:
        {
        }
        break;
    }

}

/**********************************************************
 * Application AK4642 buffer Event handler.
 * This function is called back by the AK4642 driver when
 * a AK4642 data buffer RX completes.
 **********************************************************/
void APP_AK4642MicBufferEventHandler(DRV_AK4642_BUFFER_EVENT event,
        DRV_AK4642_BUFFER_HANDLE handle, uintptr_t context )
{
    static uint8_t cnt = 0;

    switch(event)
    {
        case DRV_AK4642_BUFFER_EVENT_COMPLETE:
        {

            bufnum ^= 1;

            if(bufnum ==0)
            {
                appData.ak4642Client.rxbufferObject = (uint8_t *) micbuf1;
                appData.ak4642Client.txbufferObject = (uint8_t *) micbuf2;
            }
            else if(bufnum ==1)
            {
                appData.ak4642Client.rxbufferObject = (uint8_t *) micbuf2;
                appData.ak4642Client.txbufferObject = (uint8_t *) micbuf1;
            }

                DRV_AK4642_BufferAddWriteRead(appData.ak4642Client.handle,
                &appData.ak4642Client.writeReadBufHandle,
                appData.ak4642Client.txbufferObject,
                appData.ak4642Client.rxbufferObject,
                appData.ak4642Client.bufferSize);
                appData.state = APP_STATE_AK4642_WAIT_FOR_BUFFER_COMPLETE;
```

```
            }
            break;
            case DRV_AK4642_BUFFER_EVENT_ERROR:
            {
            } break;

            case DRV_AK4642_BUFFER_EVENT_ABORT:
            {
            } break;

        }
}

void SYS_Tasks(void)
{
    DRV_AK4642_Tasks(ak4642DevObject);
    APP_Tasks();
}
```

Configuring the Library

Macros

	Name	Description
	DRV_AK4642_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4642_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4642_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4642_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4642_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4642_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.

Description

The configuration of the AK4642 Codec Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the AK4642 Codec Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the AK4642 Codec Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

DRV_AK4642_BCLK_BIT_CLK_DIVISOR Macro

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_BCLK_BIT_CLK_DIVISOR
```

Description

AK4642 BCLK to LRCK Ratio to Generate Audio Stream

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

Following BCLK to LRCK ratios are supported 16bit data 16 bit channel :- 32fs, hence divisor would be 8 16bit data 32 bit channel :- 64fs, hence divisor would be 4

Remarks

None.

DRV_AK4642_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_CLIENTS_NUMBER DRV_AK4642_INSTANCES_NUMBER
```

Description

AK4642 Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances. Therefore, if there are five AK4642 hardware interfaces, this number will be 5.

Remarks

None.

DRV_AK4642_INPUT_REFCLOCK Macro

Identifies the input REFCLOCK source to generate the MCLK to codec.

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_INPUT_REFCLOCK
```

Description

AK4642 Input reference clock

Identifies the input REFCLOCK source to generate the MCLK to codec.

Remarks

None.

DRV_AK4642_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_INSTANCES_NUMBER
```

Description

AK4642 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of AK4642 CODEC modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_AK4642_MCLK_SAMPLE_FREQ_MULTPLIER Macro

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_MCLK_SAMPLE_FREQ_MULTPLIER
```

Description

AK4642 MCLK to LRCK Ratio to Generate Audio Stream

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency I2S sampling frequency

Supported MCLK to Sampling frequency Ratios are as below 256fs, 384fs, 512fs, 768fs or 1152fs

Remarks

None

DRV_AK4642_MCLK_SOURCE Macro

Indicate the input clock frequency to generate the MCLK to codec.

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_MCLK_SOURCE
```

Description

AK4642 Data Interface Master Clock Speed configuration

Indicate the input clock frequency to generate the MCLK to codec.

Remarks

None.

Building the Library

This section lists the files that are available in the AK4642 Codec Driver Library.

Description

This section list the files that are available in the `/src` folder of the AK4642 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/codec/ak4642`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_ak4642.h	Header file that exports the driver API.

Required File(s)

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_ak4642.c	This file contains implementation of the AK4642 Codec Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
There are no optional files for this driver.	N/A

Module Dependencies

The AK4642 Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_AK4642_Initialize	Initializes hardware and data for the instance of the AK4642 DAC module
	DRV_AK4642_Deinitialize	Deinitializes the specified instance of the AK4642 driver module
	DRV_AK4642_Status	Gets the current status of the AK4642 driver module.
	DRV_AK4642_Tasks	Maintains the driver's control and data interface state machine.

b) Client Setup Functions

	Name	Description
	DRV_AK4642_Open	Opens the specified AK4642 driver instance and returns a handle to it
	DRV_AK4642_Close	Closes an opened-instance of the AK4642 driver

c) Codec Specific Functions

	Name	Description
	DRV_AK4642_MuteOff	This function disables AK4642 output for soft mute.
	DRV_AK4642_MuteOn	This function allows AK4642 output for soft mute on.
	DRV_AK4642_SamplingRateGet	This function gets the sampling rate set on the AK4642. **Implementation:** Dynamic
	DRV_AK4642_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_AK4642_VolumeGet	This function gets the volume for AK4642 CODEC.
	DRV_AK4642_VolumeSet	This function sets the volume for AK4642 CODEC.
	DRV_AK4642_IntExtMicSet	This function sets up the codec for the internal or the external microphone use.
	DRV_AK4642_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.

d) Data Transfer Functions

	Name	Description
	DRV_AK4642_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_AK4642_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4642_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4642_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

e) Other Functions

	Name	Description
	DRV_AK4642_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.
	DRV_AK4642_VersionGet	This function returns the version of AK4642 driver
	DRV_AK4642_VersionStrGet	This function returns the version of AK4642 driver in string format.

f) Data Types and Constants

	Name	Description
	_DRV_AK4642_H	Include files.
	DRV_AK4642_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4642_COUNT	Number of valid AK4642 driver indices
	DRV_AK4642_INDEX_0	AK4642 driver index definitions
	DRV_AK4642_INDEX_1	This is macro DRV_AK4642_INDEX_1.
	DRV_AK4642_INDEX_2	This is macro DRV_AK4642_INDEX_2.
	DRV_AK4642_INDEX_3	This is macro DRV_AK4642_INDEX_3.
	DRV_AK4642_INDEX_4	This is macro DRV_AK4642_INDEX_4.
	DRV_AK4642_INDEX_5	This is macro DRV_AK4642_INDEX_5.
	DRV_AK4642_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4642_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4642_BUFFER_EVENT_HANDLER	Pointer to a AK4642 Driver Buffer Event handler function
	DRV_AK4642_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4642_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4642_COMMAND_EVENT_HANDLER	Pointer to a AK4642 Driver Command Event Handler Function
	DRV_AK4642_INIT	Defines the data required to initialize or reinitialize the AK4642 driver
	DRV_AK4642_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4642_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.

Description

This section describes the API functions of the AK4642 Codec Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_AK4642_Initialize Function

Initializes hardware and data for the instance of the AK4642 DAC module

File

drv_ak4642.h

C

```
SYS_MODULE_OBJ DRV_AK4642_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the AK4642 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Remarks

This routine must be called before any other AK4642 routine is called.

This routine should only be called once during system initialization unless DRV_AK4642_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this CODEC driver. DRV_I2C_Initialize must be called if SPI driver is used for handling the control interface of this CODEC driver.

Example

```
DRV_AK4642_INIT                init;
SYS_MODULE_OBJ                 objectHandle;

init->inUse                          = true;
init->status                         = SYS_STATUS_BUSY;
init->numClients                     = 0;
init->i2sDriverModuleIndex           = ak4642Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex           = ak4642Init->i2cDriverModuleIndex;
init->samplingRate                   = DRV_AK4642_AUDIO_SAMPLING_RATE;
init->audioDataFormat                = DRV_AK4642_AUDIO_DATA_FORMAT_MACRO;

init->isInInterruptContext           = false;

init->commandCompleteCallback = (DRV_AK4642_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;
init->mclk_multiplier = DRV_AK4642_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_AK4642_Initialize(DRV_AK4642_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_AK4642_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT *const init

);

DRV_AK4642_Deinitialize Function

Deinitializes the specified instance of the AK4642 driver module

File

drv_ak4642.h

C

```
void DRV_AK4642_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the AK4642 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_AK4642_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_AK4642_Initialize
SYS_STATUS          status;


DRV_AK4642_Deinitialize(object);

status = DRV_AK4642_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4642_Initialize routine

Function

void DRV_AK4642_Deinitialize(SYS_MODULE_OBJ object)

DRV_AK4642_Status Function

Gets the current status of the AK4642 driver module.

File

drv_ak4642.h

C

```
SYS_STATUS DRV_AK4642_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This routine provides the current status of the AK4642 driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_AK4642_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4642_Initialize
SYS_STATUS          AK4642Status;

AK4642Status = DRV_AK4642_Status(object);
if (SYS_STATUS_READY == AK4642Status)
{
    // This means the driver can be opened using the
    // DRV_AK4642_Open( ) function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4642_Initialize routine

Function

SYS_STATUS DRV_AK4642_Status(SYS_MODULE_OBJ object)

DRV_AK4642_Tasks Function

Maintains the driver's control and data interface state machine.

File

drv_ak4642.h

C

```
void DRV_AK4642_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4642_Initialize

while (true)
{
    DRV_AK4642_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4642_Initialize)

Function

void DRV_AK4642_Tasks(SYS_MODULE_OBJ object);

© 2013-2015 Microchip Technology Inc.

b) Client Setup Functions

DRV_AK4642_Open Function

Opens the specified AK4642 driver instance and returns a handle to it

File

drv_ak4642.h

C

```
DRV_HANDLE DRV_AK4642_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_AK4642_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

Description

This routine opens the specified AK4642 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

AK4642 can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_AK4642_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

Function DRV_AK4642_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_AK4642_Open(DRV_AK4642_INDEX_0, DRV_IO_INTENT_WRITEREAD | DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_AK4642_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

)

DRV_AK4642_Close Function

Closes an opened-instance of the AK4642 driver

File

drv_ak4642.h

C

```
void DRV_AK4642_Close(const DRV_HANDLE handle);
```

Returns

- None

Description

This routine closes an opened-instance of the AK4642 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_AK4642_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_AK4642_Open

DRV_AK4642_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4642_Close(DRV_Handle handle)

c) Codec Specific Functions

DRV_AK4642_MuteOff Function

This function disables AK4642 output for soft mute.

File

drv_ak4642.h

C

```
void DRV_AK4642_MuteOff(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables AK4642 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

    DRV_AK4642_MuteOff(myAK4642Handle); //AK4642 output soft mute disabled
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4642_MuteOff(DRV_HANDLE handle)

DRV_AK4642_MuteOn Function

This function allows AK4642 output for soft mute on.

File

drv_ak4642.h

C

```
void DRV_AK4642_MuteOn(DRV_HANDLE handle);
```

Returns

None.

Description

This function Enables AK4642 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

DRV_AK4642_MuteOn(myAK4642Handle);  //AK4642 output soft muted
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4642_MuteOn(DRV_HANDLE handle);

DRV_AK4642_SamplingRateGet Function

This function gets the sampling rate set on the AK4642.

Implementation: Dynamic

File

drv_ak4642.h

C

```
uint32_t DRV_AK4642_SamplingRateGet(DRV_HANDLE handle);
```

Description

This function gets the sampling rate set on the DAC AK4642.

Remarks

None.

Example

```
uint32_t baudRate;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

baudRate = DRV_AK4642_SamplingRateGet(myAK4642Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_AK4642_SamplingRateGet(DRV_HANDLE handle)

DRV_AK4642_SamplingRateSet Function

This function sets the sampling rate of the media stream.

File

drv_ak4642.h

C

```
void DRV_AK4642_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

Returns

None.

Description

This function sets the media sampling rate for the client handle.

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

DRV_AK4642_SamplingRateSet(myAK4642Handle, 48000);  //Sets 48000 media sampling rate
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
samplingRate	Sampling frequency in Hz

Function

void DRV_AK4642_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

DRV_AK4642_VolumeGet Function

This function gets the volume for AK4642 CODEC.

File

drv_ak4642.h

C

```
uint8_t DRV_AK4642_VolumeGet(DRV_HANDLE handle, DRV_AK4642_CHANNEL channel);
```

Returns

None.

Description

This functions gets the current volume programmed to the CODEC AK4642.

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

  volume = DRV_AK4642_VolumeGet(myAK4642Handle, DRV_AK4642_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified

Function

uint8_t DRV_AK4642_VolumeGet(DRV_HANDLE handle, DRV_AK4642_CHANNEL channel)

DRV_AK4642_VolumeSet Function

This function sets the volume for AK4642 CODEC.

File

drv_ak4642.h

C

```
void DRV_AK4642_VolumeSet(DRV_HANDLE handle, DRV_AK4642_CHANNEL channel, uint8_t volume);
```

Returns

None

Description

This functions sets the volume value from 0-255. The codec has DAC value to volume range mapping as :- 00 H : +12dB FF H : -115dB In order to make the volume value to dB mapping monotonically increasing from 00 to FF, re-mapping is introduced which reverses the volume value to dB mapping as well as normalizes the volume range to a more audible dB range. The current driver implementation assumes that all dB values under -60 dB are inaudible to the human ear. Re-Mapped values 00 H : -60 dB FF H : +12 dB

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

    DRV_AK4642_VolumeSet(myAK4642Handle,DRV_AK4642_CHANNEL_LEFT, 120);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified
volume	volume value specified in the range 0-255 (0x00 to 0xFF)

Function

void DRV_AK4642_VolumeSet(DRV_HANDLE handle, DRV_AK4642_CHANNEL channel, uint8_t volume);

DRV_AK4642_IntExtMicSet Function

This function sets up the codec for the internal or the external microphone use.

File

drv_ak4642.h

C

```
void DRV_AK4642_IntExtMicSet(DRV_HANDLE handle, DRV_AK4642_INT_EXT_MIC micInput);
```

Returns

None

Description

This function sets up the codec for the internal or the external microphone use.

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

micInput	Internal vs External mic input

Function

void DRV_AK4642_IntExtMicSet(DRV_HANDLE handle);

DRV_AK4642_MonoStereoMicSet Function

This function sets up the codec for the Mono or Stereo microphone mode.

File

drv_ak4642.h

C

```
void DRV_AK4642_MonoStereoMicSet(DRV_HANDLE handle, DRV_AK4642_MONO_STEREO_MIC mono_stereo_mic);
```

Returns

None

Description

This function sets up the codec for the Mono or Stereo microphone mode.

Remarks

Currently the ak4642 codec does not work in the MONO_LEFT_CHANNEL mode. This issue will be followed up with AKM.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
mono_stereo_mic	Mono / Stereo mic setup

Function

void DRV_AK4642_MonoStereoMicSet(DRV_HANDLE handle);

d) Data Transfer Functions

DRV_AK4642_BufferAddWrite Function

Schedule a non-blocking driver write operation.

File

drv_ak4642.h

C

```
void DRV_AK4642_BufferAddWrite(const DRV_HANDLE handle, DRV_AK4642_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4642_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4642_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4642_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4642_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4642 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4642 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 device instance and the DRV_AK4642_Status must have returned SYS_STATUS_READY.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK4642_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4642_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

// Client registers an event handler with driver

DRV_AK4642_BufferEventHandlerSet(myAK4642Handle,
                APP_AK4642BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4642_BufferAddWrite(myAK4642handle, &bufferHandle
```

```
                                                myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4642_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4642BufferEventHandler(DRV_AK4642_BUFFER_EVENT event,
        DRV_AK4642_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4642_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4642_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4642 instance as return by the DRV_AK4642_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4642_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_AK4642_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

DRV_AK4642_BufferAddRead Function

Schedule a non-blocking driver read operation.

File

drv_ak4642.h

C

```
void DRV_AK4642_BufferAddRead(const DRV_HANDLE handle, DRV_AK4642_BUFFER_HANDLE * bufferHandle,
void * buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4642_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4642_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4642_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4642_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4642 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4642 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 device instance and the DRV_AK4642_Status must have returned SYS_STATUS_READY.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_AK4642_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4642 instance as return by the DRV_AK4642_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

```
void DRV_AK4642_BufferAddRead
(
const        DRV_HANDLE handle,
        DRV_AK4642_BUFFER_HANDLE *bufferHandle,
void *buffer, size_t size
)
```

DRV_AK4642_BufferAddWriteRead Function

Schedule a non-blocking driver write-read operation.

Implementation: Dynamic

File

drv_ak4642.h

C

```
void DRV_AK4642_BufferAddWriteRead(const DRV_HANDLE handle, DRV_AK4642_BUFFER_HANDLE *
bufferHandle, void * transmitBuffer, void * receiveBuffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4642_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4642_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4642_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4642_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4642 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4642 driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every AK4642 write. The transmit and receive size must be same.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 device instance and the DRV_AK4642_Status must have returned SYS_STATUS_READY.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_AK4642_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_AK4642_BUFFER_HANDLE bufferHandle;

// myak4642Handle is the handle returned
// by the DRV_AK4642_Open function.

// Client registers an event handler with driver

DRV_AK4642_BufferEventHandlerSet(myak4642Handle,
                APP_AK4642BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4642_BufferAddWriteRead(myak4642handle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_AK4642_BUFFER_HANDLE_INVALID == bufferHandle)
{
```

```
        // Error handling here
    }

    // Event is received when
    // the buffer is processed.

    void APP_AK4642BufferEventHandler(DRV_AK4642_BUFFER_EVENT event,
            DRV_AK4642_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
    {
        // contextHandle points to myAppObj.

        switch(event)
        {
            case DRV_AK4642_BUFFER_EVENT_COMPLETE:

                // This means the data was transferred.
                break;

            case DRV_AK4642_BUFFER_EVENT_ERROR:

                // Error handling here.
                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	Handle of the AK4642 instance as returned by the DRV_AK4642_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	The buffer where the transmit data will be stored
receiveBuffer	The buffer where the received data will be stored
size	Buffer size in bytes

Function

void DRV_AK4642_BufferAddWriteRead

(

const DRV_HANDLE handle,

DRV_AK4642_BUFFER_HANDLE *bufferHandle,

void *transmitBuffer,

void *receiveBuffer,

size_t size

)

DRV_AK4642_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

File

drv_ak4642.h

C

```
void DRV_AK4642_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK4642_BUFFER_EVENT_HANDLER
```

```
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_AK4642_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4642_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

// Client registers an event handler with driver

DRV_AK4642_BufferEventHandlerSet(myAK4642Handle,
                APP_AK4642BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4642_BufferAddWrite(myAK4642handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4642_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4642BufferEventHandler(DRV_AK4642_BUFFER_EVENT event,
        DRV_AK4642_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4642_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4642_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;
```

```
        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

```
void DRV_AK4642_BufferEventHandlerSet
(
    DRV_HANDLE handle,
const    DRV_AK4642_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t contextHandle
)
```

e) Other Functions

DRV_AK4642_CommandEventHandlerSet Function

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

File

drv_ak4642.h

C

```
void DRV_AK4642_CommandEventHandlerSet(DRV_HANDLE handle, const
DRV_AK4642_COMMAND_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK4642_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "AK4642 CODEC Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4642_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

// Client registers an event handler with driver

DRV_AK4642_CommandEventHandlerSet(myAK4642Handle,
                APP_AK4642CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK4642_DeEmphasisFilterSet(myAK4642Handle, DRV_AK4642_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK4642CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
```

```
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4642_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4642_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4642_VersionGet Function

This function returns the version of AK4642 driver

File

drv_ak4642.h

C

```
uint32_t DRV_AK4642_VersionGet();
```

Returns

returns the version of AK4642 driver.

Description

The version number returned from the DRV_AK4642_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Remarks

None.

Preconditions

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
uint32_t AK4642version;
AK4642version = DRV_AK4642_VersionGet();
```

Function

uint32_t DRV_AK4642_VersionGet(void)

DRV_AK4642_VersionStrGet Function

This function returns the version of AK4642 driver in string format.

File

drv_ak4642.h

C

```
int8_t* DRV_AK4642_VersionStrGet();
```

Returns

returns a string containing the version of AK4642 driver.

Description

The DRV_AK4642_VersionStrGet function returns a string in the format: ".[.][]" Where: is the AK4642 driver's version number. is the AK4642 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces. For example, "0.03a" "1.00"

Remarks

None

Preconditions

None.

Example

```
int8_t *AK4642string;
AK4642string = DRV_AK4642_VersionStrGet();
```

Function

int8_t* DRV_AK4642_VersionStrGet(void)

f) Data Types and Constants

_DRV_AK4642_H Macro

File

drv_ak4642.h

C

```
#define _DRV_AK4642_H
```

Description

Include files.

DRV_AK4642_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_ak4642.h

C

```
#define DRV_AK4642_BUFFER_HANDLE_INVALID ((DRV_AK4642_BUFFER_HANDLE)(-1))
```

Description

AK4642 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_AK4642_BufferAddWrite() and the DRV_AK4642_BufferAddRead() function if the buffer add request was not successful.

Remarks

None.

DRV_AK4642_COUNT Macro

Number of valid AK4642 driver indices

File

drv_ak4642.h

C

```
#define DRV_AK4642_COUNT
```

Description

AK4642 Driver Module Count

This constant identifies the maximum number of AK4642 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of AK4642 instances on this microcontroller.

Remarks

This value is part-specific.

DRV_AK4642_INDEX_0 Macro

AK4642 driver index definitions

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_0 0
```

Description

Driver AK4642 Module Index

These constants provide AK4642 driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_AK4642_Initialize and DRV_AK4642_Open routines to identify the driver instance in use.

DRV_AK4642_INDEX_1 Macro

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_1 1
```

Description

This is macro DRV_AK4642_INDEX_1.

DRV_AK4642_INDEX_2 Macro

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_2 2
```

Description

This is macro DRV_AK4642_INDEX_2.

DRV_AK4642_INDEX_3 Macro

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_3 3
```

Description

This is macro DRV_AK4642_INDEX_3.

DRV_AK4642_INDEX_4 Macro

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_4 4
```

Description

This is macro DRV_AK4642_INDEX_4.

DRV_AK4642_INDEX_5 Macro

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_5 5
```

Description

This is macro DRV_AK4642_INDEX_5.

DRV_AK4642_AUDIO_DATA_FORMAT Enumeration

Identifies the Serial Audio data interface format.

File

drv_ak4642.h

C

```
typedef enum {
  DRV_AK4642_AUDIO_DATA_FORMAT_NOT_APPLICABLE = 0,
  DRV_AK4642_AUDIO_DATA_FORMAT_16BITMSB_SDTO_16BITLSB_SDTI,
  DRV_AK4642_AUDIO_DATA_FORMAT_16BITMSB_SDTO_16BITMSB_SDTI,
  DRV_AK4642_AUDIO_DATA_FORMAT_I2S
} DRV_AK4642_AUDIO_DATA_FORMAT;
```

Description

AK4642 Audio data format

This enumeration identifies Serial Audio data interface format.

DRV_AK4642_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_ak4642.h

C

```
typedef enum {
  DRV_AK4642_BUFFER_EVENT_COMPLETE,
  DRV_AK4642_BUFFER_EVENT_ERROR,
  DRV_AK4642_BUFFER_EVENT_ABORT
} DRV_AK4642_BUFFER_EVENT;
```

Members

Members	Description
DRV_AK4642_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_AK4642_BUFFER_EVENT_ERROR	Error while processing the request
DRV_AK4642_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

AK4642 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_AK4642_BufferAddWrite() or the DRV_AK4642_BufferAddRead() function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_AK4642_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_AK4642_BUFFER_EVENT_HANDLER Type

Pointer to a AK4642 Driver Buffer Event handler function

File

drv_ak4642.h

C

```
typedef void (* DRV_AK4642_BUFFER_EVENT_HANDLER)(DRV_AK4642_BUFFER_EVENT event,
DRV_AK4642_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

AK4642 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK4642 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_AK4642_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK4642_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_AK4642_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was

registered using the DRV_AK4642_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver(i2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_AK4642_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_AK4642_BUFFER_EVENT event,
                               DRV_AK4642_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK4642_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK4642_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4642_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

File

drv_ak4642.h

C

```
typedef uintptr_t DRV_AK4642_BUFFER_HANDLE;
```

Description

AK4642 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK4642_BufferAddWrite() or DRV_AK4642_BufferAddRead() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_AK4642_CHANNEL Enumeration

Identifies Left/Right Audio channel

File

drv_ak4642.h

C

```
typedef enum {
  DRV_AK4642_CHANNEL_LEFT,
  DRV_AK4642_CHANNEL_RIGHT,
  DRV_AK4642_CHANNEL_LEFT_RIGHT,
  DRV_AK4642_NUMBER_OF_CHANNELS
} DRV_AK4642_CHANNEL;
```

Description

AK4642 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

DRV_AK4642_COMMAND_EVENT_HANDLER Type

Pointer to a AK4642 Driver Command Event Handler Function

File

drv_ak4642.h

C

```
typedef void (* DRV_AK4642_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

Returns

None.

Description

AK4642 Driver Command Event Handler Function

This data type defines the required function signature for the AK4642 driver command event handling callback function.

A command is a control instruction to the AK4642 CODEC. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4642_CommandEventHandlerSet function. This context handle value is passed back to

the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_AK4642CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4642_INIT Structure

Defines the data required to initialize or reinitialize the AK4642 driver

File

drv_ak4642.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX i2sDriverModuleIndex;
  SYS_MODULE_INDEX i2cDriverModuleIndex;
  uint32_t samplingRate;
  uint8_t volume;
  DRV_AK4642_AUDIO_DATA_FORMAT audioDataFormat;
} DRV_AK4642_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX i2sDriverModuleIndex;	Identifies data module(I2S) driver ID for data interface of CODEC
SYS_MODULE_INDEX i2cDriverModuleIndex;	Identifies data module(I2C) driver ID for control interface of CODEC
uint32_t samplingRate;	Sampling rate
uint8_t volume;	Volume
DRV_AK4642_AUDIO_DATA_FORMAT audioDataFormat;	Identifies the Audio data format

Description

AK4642 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK4642 CODEC driver.

Remarks

None.

DRV_AK4642_INT_EXT_MIC Enumeration

Identifies the Mic input source.

File

drv_ak4642.h

C

```
typedef enum {
  INT_MIC,
  EXT_MIC
} DRV_AK4642_INT_EXT_MIC;
```

Description

AK4642 Mic Internal / External Input

This enumeration identifies the Mic input source.

DRV_AK4642_MONO_STEREO_MIC Enumeration

Identifies the Mic input as Mono / Stereo.

File

drv_ak4642.h

C

```
typedef enum {
  ALL_ZEROS,
  MONO_RIGHT_CHANNEL,
  MONO_LEFT_CHANNEL,
  STEREO
} DRV_AK4642_MONO_STEREO_MIC;
```

Description

AK4642 Mic Mono / Stereo Input

This enumeration identifies the Mic input as Mono / Stereo.

Files

Files

Name	Description
drv_ak4642.h	AK4642 CODEC Driver Interface header file
drv_ak4642_config_template.h	AK4642 Codec Driver Configuration Template.

Description

This section lists the source and header files used by the AK4642 Codec Driver Library.

drv_ak4642.h

AK4642 CODEC Driver Interface header file

Enumerations

	Name	Description
	DRV_AK4642_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4642_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4642_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4642_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4642_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.

Functions

	Name	Description
	DRV_AK4642_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4642_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_AK4642_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4642_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_AK4642_Close	Closes an opened-instance of the AK4642 driver
	DRV_AK4642_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.
	DRV_AK4642_Deinitialize	Deinitializes the specified instance of the AK4642 driver module
	DRV_AK4642_Initialize	Initializes hardware and data for the instance of the AK4642 DAC module
	DRV_AK4642_IntExtMicSet	This function sets up the codec for the internal or the external microphone use.
	DRV_AK4642_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4642_MuteOff	This function disables AK4642 output for soft mute.
	DRV_AK4642_MuteOn	This function allows AK4642 output for soft mute on.
	DRV_AK4642_Open	Opens the specified AK4642 driver instance and returns a handle to it
	DRV_AK4642_SamplingRateGet	This function gets the sampling rate set on the AK4642. **Implementation:** Dynamic
	DRV_AK4642_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_AK4642_Status	Gets the current status of the AK4642 driver module.
	DRV_AK4642_Tasks	Maintains the driver's control and data interface state machine.
	DRV_AK4642_VersionGet	This function returns the version of AK4642 driver
	DRV_AK4642_VersionStrGet	This function returns the version of AK4642 driver in string format.
	DRV_AK4642_VolumeGet	This function gets the volume for AK4642 CODEC.
	DRV_AK4642_VolumeSet	This function sets the volume for AK4642 CODEC.

Macros

	Name	Description
	_DRV_AK4642_H	Include files.
	DRV_AK4642_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4642_COUNT	Number of valid AK4642 driver indices
	DRV_AK4642_INDEX_0	AK4642 driver index definitions
	DRV_AK4642_INDEX_1	This is macro DRV_AK4642_INDEX_1.
	DRV_AK4642_INDEX_2	This is macro DRV_AK4642_INDEX_2.
	DRV_AK4642_INDEX_3	This is macro DRV_AK4642_INDEX_3.
	DRV_AK4642_INDEX_4	This is macro DRV_AK4642_INDEX_4.
	DRV_AK4642_INDEX_5	This is macro DRV_AK4642_INDEX_5.

Structures

	Name	Description
	DRV_AK4642_INIT	Defines the data required to initialize or reinitialize the AK4642 driver

Types

	Name	Description
	DRV_AK4642_BUFFER_EVENT_HANDLER	Pointer to a AK4642 Driver Buffer Event handler function
	DRV_AK4642_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4642_COMMAND_EVENT_HANDLER	Pointer to a AK4642 Driver Command Event Handler Function

Description

AK4642 CODEC Driver Interface

The AK4642 CODEC device driver interface provides a simple interface to manage the AK4642 16/24-Bit CODEC that can be interfaced Microchip Microcontroller. This file provides the interface definition for the AK4642 CODEC device driver.

File Name

drv_ak4642.h

Company

Microchip Technology Inc.

drv_ak4642_config_template.h

AK4642 Codec Driver Configuration Template.

Macros

	Name	Description
	DRV_AK4642_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4642_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4642_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4642_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4642_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4642_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.

Description

AK4642 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_ak4642_config_template.h

Company

Microchip Technology Inc.

AK4645 Codec Driver Library

This topic describes the AK4645 Codec Driver Library.

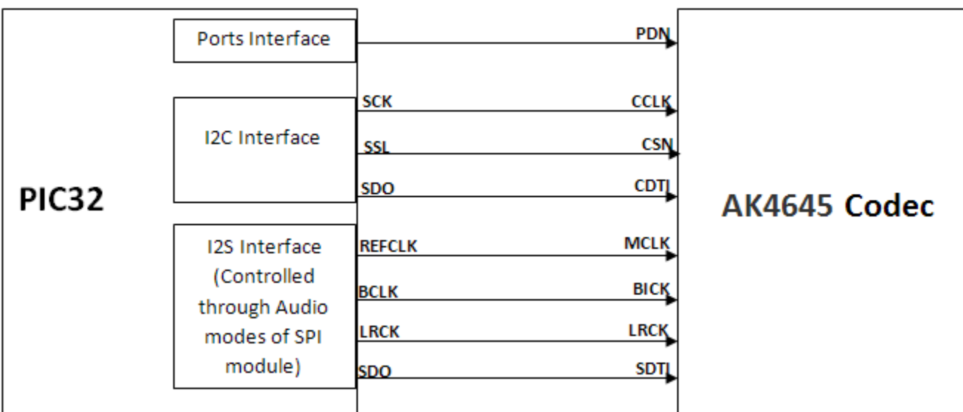
Introduction

This library provides an interface to manage the AK4645 Codec that is serially interfaced to a Microchip microcontroller for providing Audio Solutions.

Description

The AK4645 module is 16/24-bit Audio Codec from Asahi Kasei Microdevices Corporation. The AK4645 can be interfaced to Microchip microcontrollers through I2C and I2S serial interfaces. The I2C interface is used for control command transfer. The I2S interface is used for Audio data output.

A typical interface of AK4645 to a Microchip PIC32 device is provided in the following diagram:



Features

The AK4645 Codec Driver supports the following features:

- Audio Interface Format: MSB first
- ADC: 16-bit MSB justified, I2S, DSP modes
- DAC: 16-bit MSB justified, 16-bit LSB justified, 16/24-bit I2S, DSP modes
- Sampling Frequency Range: 8 kHz to 48 kHz
- Digital Volume Control: +12dB ~ .115dB, 0.5dB Step
- SoftMute: On and Off
- Master Clock Frequencies: 32 fs/64 fs/128fs/256fs

Using the Library

This topic describes the basic architecture of the AK4645 Codec Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_ak4645.h`

The interface to the AK4645 Codec Driver library is defined in the `drv_ak4645.h` header file. Any C language source (`.c`) file that uses the AK4645 Codec Driver library should include this header.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

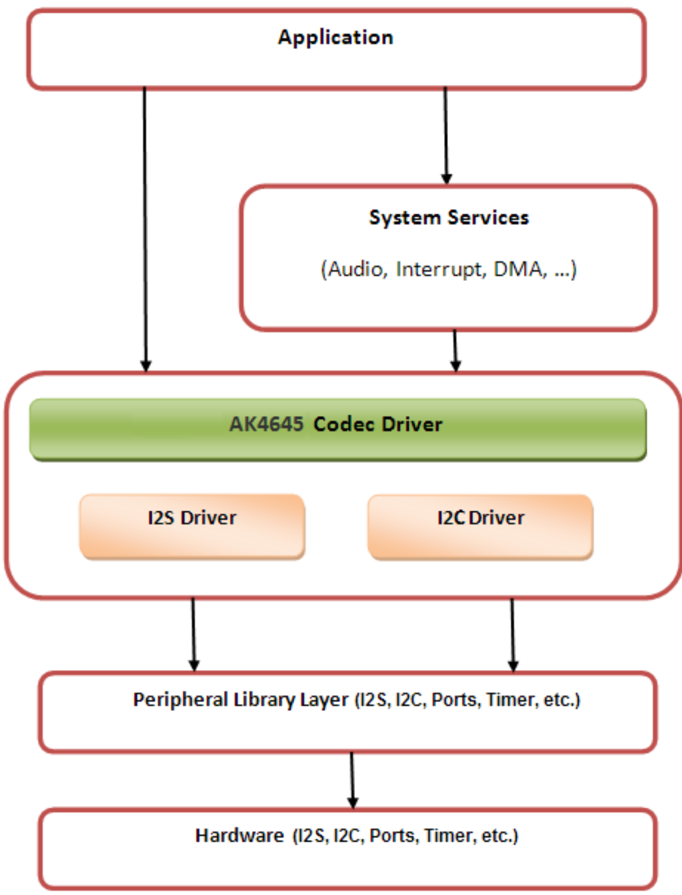
Abstraction Model

This library provides a low-level abstraction of the AK4645 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the AK4645 Codec Driver is positioned in the MPLAB Harmony framework. The AK4645 Codec Driver uses the SPI and I2S drivers for control and audio data transfers to the AK4645 module.

AK4645 Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The AK4645 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK4645 DAC module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK4645 Codec Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open and close functions.
Codec Specific Functions	Provides functions that are codec specific.
Data Transfer Functions	Provides data transfer functions.
Other Functions	Provides driver specific miscellaneous functions such as sampling rate setting, control command functions, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the AK4645 Codec Driver Library.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

System Access

This topic provides information on system initialization, implementations, and provides a system access code example.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the AK4645 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK4645_INIT or by using Initialization Overrides) that are supported by the specific AK4645 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver.
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver.
- Sampling rate
- Master clock detection mode
- Power down pin port initialization

The DRV_AK4645_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ AK4645_Deinitialize, DRV_ AK4645_Status and DRV_I2S_Tasks.

Implementations

The implementation of the AK4645 Codec Driver has dedicated hardware for control (I2C) and data (I2S) interface with the standard MPLAB Harmony I2C and I2S driver interfaces.

Example:

```
DRV_AK4645_INIT drvak4645Init =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .i2sDriverModuleIndex = DRV_AK4645_I2S_DRIVER_MODULE_INDEX_IDX0,
    .i2cDriverModuleIndex = DRV_AK4645_I2C_DRIVER_MODULE_INDEX_IDX0,
    .volume = DRV_AK4645_VOLUME,
};

/*
  The I2C and I2S module index should be same as the one used in
  initializing the I2C and I2S drivers.
 */

ak4645DevObject = DRV_AK4645_Initialize(DRV_AK4645_INDEX_0, (SYS_MODULE_INIT *) &drvak4645Init);
if (SYS_MODULE_OBJ_INVALID == ak4645DevObject)
{
    // Handle error
}
```

Task Routine

The DRV_AK4645_Tasks will be called from the System Task Service.

Client Access

This topic describes client access and includes a code example.

Description

For the application to start using an instance of the module, it must call the DRV_AK4645_Open function. The DRV_AK4645_Open provides a driver handle to the AK4645 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK4645_Deinitialize, the application must call the DRV_AK4645_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Note: It is necessary to check the status of driver initialization before opening a driver instance. The status of the AK4645 Codec Driver can be known by calling DRV_AK4645_Status.

Example:

```
DRV_HANDLE handle;
SYS_STATUS ak4645Status;
    ak4645Status = DRV_AK4645_Status(sysObjects.ak4645DevObject);
        if (SYS_STATUS_READY == ak4645Status)
        {
               // The driver can now be opened.
               appData.ak4645Client.handle = DRV_AK4645_Open
                                             (DRV_AK4645_INDEX_0,
                                              DRV_IO_INTENT_WRITE |
                                              DRV_IO_INTENT_EXCLUSIVE);
               if(appData.ak4645Client.handle != DRV_HANDLE_INVALID)
               {
                   appData.state = APP_STATE_AK4645_SET_BUFFER_HANDLER;
               }
               else
               {
                   SYS_DEBUG(0, "Find out what's wrong \r\n");
               }
    }
    else
    {
        /* AK4645 Driver Is not ready */
                ;
    }
```

Client Operations

This topic describes client operations and provides a code example.

Description

Client operations provide the API interface for control command and audio data transfer to the AK4645 Codec.

The following AK4645 Codec specific control command functions are provided:

- DRV_AK4645_SamplingRateSet
- DRV_AK4645_SamplingRateGet
- DRV_AK4645_VolumeSet
- DRV_AK4645_VolumeGet
- DRV_AK4645_MuteOn
- DRV_AK4645_MuteOff

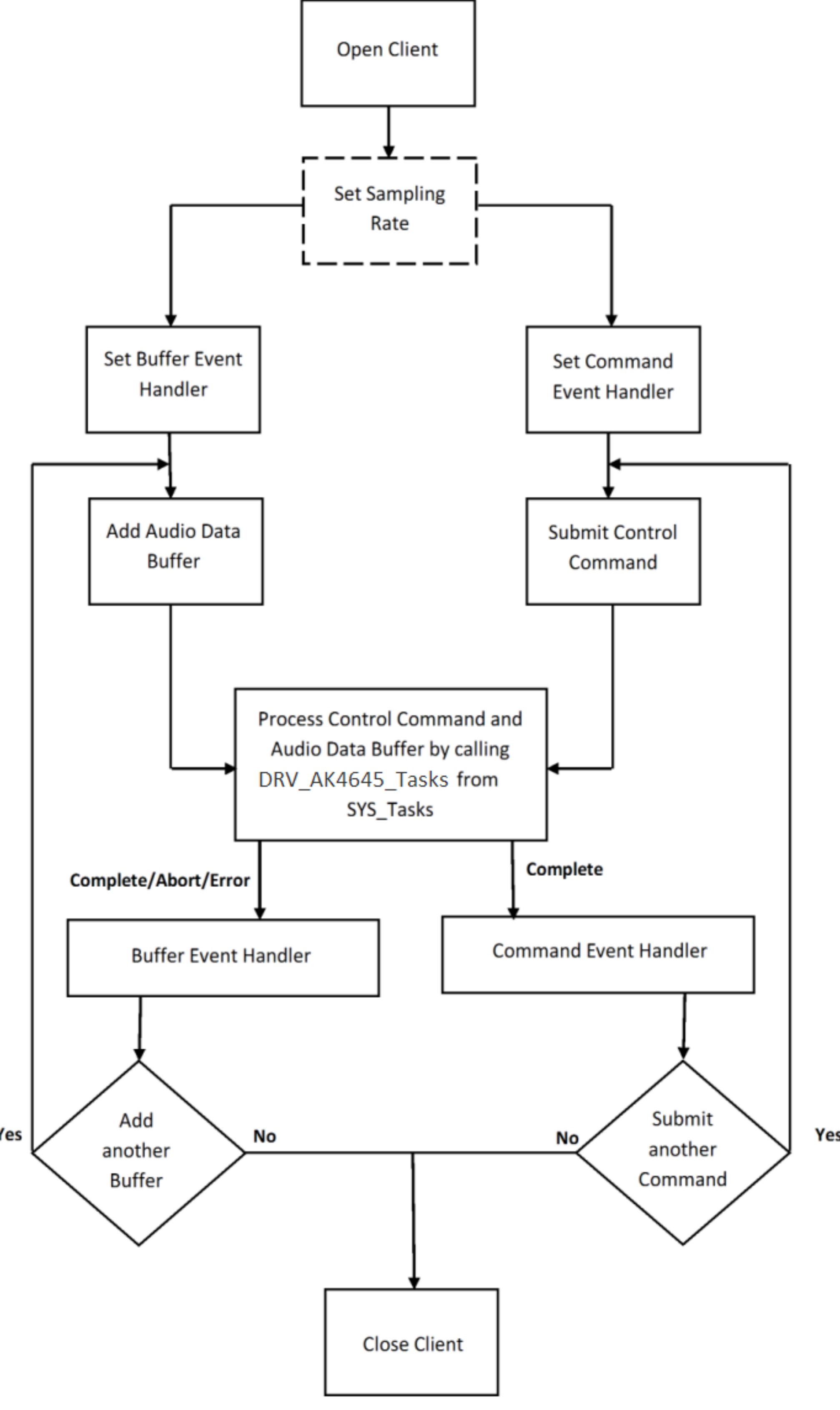
These functions schedule a non-blocking control command transfer operation. These functions submit the control

command request to the AK4645 Codec. A notification for the submitted requests can be received by registering a command callback event with the driver. The driver notifies by calling the callback on successfully transmitting the command to the AK4645 Codec module.

DRV_AK4645_BufferAddWrite, DRV_AK4645_BufferAddRead, and DRV_AK4645_BufferAddWriteRead are buffered data operation functions. These functions schedule non-blocking audio data transfer operations. The function adds the request to the hardware instance queues and returns a buffer handle. The requesting client also registers a callback event with the driver. The driver notifies the client with DRV_AK4645_BUFFER_EVENT_COMPLETE, DRV_AK4645_BUFFER_EVENT_ERROR, or DRV_AK4645_BUFFER_EVENT_ABORT events.

The submitted control commands and audio buffer add requests are processed under DRV_AK4645_Tasks function. This function is called from the SYS_Tasks routine.

The following diagram illustrates the control commands and audio buffered data operations.



Note: It is not necessary to close and reopen the client between multiple transfers.

An application using the buffered functionality needs to perform the following steps:

1. The system should have completed necessary setup and initializations.

2. The I2S driver object should have been initialized by calling DRV_I2S_Initialize.
3. The I2C driver object should have been initialized by calling DRV_I2C_Initialize.
4. The AK4645 driver object should be initialized by calling DRV_AK4645_Initialize.
5. The necessary sampling rate value should be set up by calling DRV_AK4645_ SamplingRateSet.
6. Register buffer event handler for the client handle by calling DRV_AK4645_BufferEventHandlerSet.
7. Register command event handler for the client handle by calling DRV_AK4645_CommandEventHandlerSet.
8. Submit a command by calling specific command API.
9. Add a buffer to initiate the data transfer by calling DRV_AK4645_BufferAddWrite, DRV_AK4645_BufferAddRead, and DRV_AK4645_BufferAddWriteRead.
10. The submitted command and Audio data processing happens b calling DRV_AK4645_Tasks from SYS_Tasks.
11. Repeat steps 9 through 10 to handle multiple buffer transmission and reception.
12. When the client is done, it can use DRV_AK4645_Close to close the client handle.

Example:

```
typedef enum
{
    APP_STATE_AK4645_OPEN,
    APP_STATE_AK4645_SET_BUFFER_HANDLER,
    APP_STATE_AK4645_ADD_FIRST_BUFFER_READ,
    APP_STATE_AK4645_ADD_BUFFER_OUT,
    APP_STATE_AK4645_ADD_BUFFER_IN,
    APP_STATE_AK4645_WAIT_FOR_BUFFER_COMPLETE,
} APP_STATES;

typedef struct
{
    DRV_HANDLE handle;
    DRV_AK4645_BUFFER_HANDLE writereadBufHandle;
    DRV_AK4645_BUFFER_EVENT_HANDLER bufferEventHandler;
    uintptr_t context;
    uint8_t *txbufferObject;
    uint8_t *rxbufferObject;
    size_t bufferSize;

} APP_AK4645_CLIENT;

typedef struct
{
    /* Application's current state*/
    APP_STATES state;
    /* USART client handle */
    APP_AK4645_CLIENT ak4645Client;
} APP_DATA;
APP_DATA appData;
SYS_MODULE_OBJ ak4645DevObject;
DRV_AK4645_INIT drvak4645Init =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .i2sDriverModuleIndex = DRV_AK4645_I2S_DRIVER_MODULE_INDEX_IDX0,
    .i2cDriverModuleIndex = DRV_AK4645_I2C_DRIVER_MODULE_INDEX_IDX0,
    .volume = DRV_AK4645_VOLUME,
};

void SYS_Initialize(void * data)
{
    /* Initialize Drivers */
    DRV_I2C0_Initialize();
    sysObj.drvI2S0 = DRV_I2S_Initialize(DRV_I2S_INDEX_0, (SYS_MODULE_INIT *)
                                        &drvI2S0InitData);

    sysObj.drvak4645Codec0 = DRV_AK4645_Initialize(DRV_AK4645_INDEX_0,
                             (SYS_MODULE_INIT *)&drvak4645Codec0InitData);

    /* Initialize System Services */
    SYS_INT_Initialize();
```

```
}

void APP_Tasks (void )
{
    switch(appData.state)
    {
        case APP_STATE_AK4645_OPEN:
        {
            /* A client opens the driver object to get an Handle */
            appData.ak4645Client.handle = DRV_AK4645_Open(DRV_AK4645_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
            if(appData.ak4645Client.handle != DRV_HANDLE_INVALID)
            {
                appData.state = APP_STATE_AK4645_SET_BUFFER_HANDLER;
            }
            else
            {
                /* Got an Invalid Handle.  Wait for AK4645 to Initialize */
                ;
            }
        }
        break;

        /* Set a handler for the audio buffer completion event */
        case APP_STATE_AK4645_SET_BUFFER_HANDLER:
        {

           DRV_AK4645_BufferEventHandlerSet(appData.ak4645Client.handle,
                    appData.ak4645Client.bufferEventHandler,
                    appData.ak4645Client.context);

            appData.state = APP_STATE_AK4645_ADD_FIRST_BUFFER_READ;
        }
        break;

        case APP_STATE_AK4645_ADD_FIRST_BUFFER_READ:
        {
            DRV_AK4645_BufferAddWriteRead(appData.ak4645Client.handle,
                    &appData.ak4645Client.writeReadBufHandle,
                    appData.ak4645Client.txbufferObject,
                    appData.ak4645Client.rxbufferObject,
                    appData.ak4645Client.bufferSize);
            if(appData.ak4645Client.writeReadBufHandle != DRV_AK4645_BUFFER_HANDLE_INVALID)
            {
                appData.state = APP_STATE_AK4645_WAIT_FOR_BUFFER_COMPLETE;
            }
            else
            {
                SYS_DEBUG(0, "Find out what is wrong \r\n");
            }

        }
        break;
        /* Add an audio buffer to the ak4645 driver to be transmitted to
         * AK4645 CODEC */
        case APP_STATE_AK4645_ADD_BUFFER_OUT:
        {

            DRV_AK4645_BufferAddWrite(appData.ak4645Client.handle, &appData.ak4645Client.writeBufHandle,
            appData.ak4645Client.txbufferObject, appData.ak4645Client.bufferSize);
            if(appData.ak4645Client.writeBufHandle != DRV_AK4645_BUFFER_HANDLE_INVALID)
            {

                appData.state = APP_STATE_AK4645_WAIT_FOR_BUFFER_COMPLETE;
            }
            else
            {
                SYS_DEBUG(0, "Find out what is wrong \r\n");
            }
        }
```

```
        break;
        /* Add an audio buffer to the ak4645 driver to be received
         * AK4645 CODEC */
        case APP_STATE_AK4645_ADD_BUFFER_IN:
        {

            DRV_AK4645_BufferAddRead(appData.ak4645Client.handle, &appData.ak4645Client.readBufHandle,
            appData.ak4645Client.rxbufferObject, appData.ak4645Client.bufferSize);


            if(appData.ak4645Client.readBufHandle != DRV_AK4645_BUFFER_HANDLE_INVALID)
            {
                appData.state = APP_STATE_AK4645_ADD_BUFFER_OUT;
            }
            else
            {
                SYS_DEBUG(0, "Find out what is wrong \r\n");
            }
        }
        break;
        /* Audio data Transmission under process */
        case APP_STATE_AK4645_WAIT_FOR_BUFFER_COMPLETE:
        {
            /*Do nothing*/
        }
        break;

        default:
        {
        }
        break;
    }

}

/**********************************************************
 * Application AK4645 buffer Event handler.
 * This function is called back by the AK4645 driver when
 * a AK4645 data buffer RX completes.
 **********************************************************/
void APP_AK4645MicBufferEventHandler(DRV_AK4645_BUFFER_EVENT event,
        DRV_AK4645_BUFFER_HANDLE handle, uintptr_t context )
{
    static uint8_t cnt = 0;

    switch(event)
    {
        case DRV_AK4645_BUFFER_EVENT_COMPLETE:
        {

            bufnum ^= 1;

            if(bufnum ==0)
            {
                appData.ak4645Client.rxbufferObject = (uint8_t *) micbuf1;
                appData.ak4645Client.txbufferObject = (uint8_t *) micbuf2;
            }
            else if(bufnum ==1)
            {
                appData.ak4645Client.rxbufferObject = (uint8_t *) micbuf2;
                appData.ak4645Client.txbufferObject = (uint8_t *) micbuf1;
            }

                DRV_AK4645_BufferAddWriteRead(appData.ak4645Client.handle,
                &appData.ak4645Client.writeReadBufHandle,
                appData.ak4645Client.txbufferObject,
                appData.ak4645Client.rxbufferObject,
                appData.ak4645Client.bufferSize);
                appData.state = APP_STATE_AK4645_WAIT_FOR_BUFFER_COMPLETE;
```

```
            }
            break;
            case DRV_AK4645_BUFFER_EVENT_ERROR:
            {
            } break;

            case DRV_AK4645_BUFFER_EVENT_ABORT:
            {
            } break;

        }
}

void SYS_Tasks(void)
{
    DRV_AK4645_Tasks(ak4645DevObject);
    APP_Tasks();
}
```

Configuring the Library

Macros

	Name	Description
	DRV_AK4645_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4645_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4645_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4645_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4645_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4645_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.

Description

The configuration of the AK4645 Codec Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the AK4645 Codec Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the AK4645 Codec Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

DRV_AK4645_BCLK_BIT_CLK_DIVISOR Macro

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

File

drv_ak4645_config_template.h

C

```
#define DRV_AK4645_BCLK_BIT_CLK_DIVISOR
```

Description

AK4645 BCLK to LRCK Ratio to Generate Audio Stream

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

Following BCLK to LRCK ratios are supported 16bit data 16 bit channel :- 32fs, hence divisor would be 8 16bit data 32 bit channel :- 64fs, hence divisor would be 4

Remarks

None.

DRV_AK4645_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_ak4645_config_template.h

C

```
#define DRV_AK4645_CLIENTS_NUMBER DRV_AK4645_INSTANCES_NUMBER
```

Description

AK4645 Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances. Therefore, if there are five AK4645 hardware interfaces, this number will be 5.

Remarks

None.

DRV_AK4645_INPUT_REFCLOCK Macro

Identifies the input REFCLOCK source to generate the MCLK to codec.

File

drv_ak4645_config_template.h

C

```
#define DRV_AK4645_INPUT_REFCLOCK
```

Description

AK4645 Input reference clock

Identifies the input REFCLOCK source to generate the MCLK to codec.

Remarks

None.

DRV_AK4645_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_ak4645_config_template.h

C

```
#define DRV_AK4645_INSTANCES_NUMBER
```

Description

AK4645 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of AK4645 CODEC modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_AK4645_MCLK_SAMPLE_FREQ_MULTPLIER Macro

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

File

drv_ak4645_config_template.h

C

```
#define DRV_AK4645_MCLK_SAMPLE_FREQ_MULTPLIER
```

Description

AK4645 MCLK to LRCK Ratio to Generate Audio Stream

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency I2S sampling frequency

Supported MCLK to Sampling frequency Ratios are as below 256fs, 384fs, 512fs, 768fs or 1152fs

Remarks

None

DRV_AK4645_MCLK_SOURCE Macro

Indicate the input clock frequency to generate the MCLK to codec.

File

drv_ak4645_config_template.h

C

```
#define DRV_AK4645_MCLK_SOURCE
```

Description

AK4645 Data Interface Master Clock Speed configuration

Indicate the input clock frequency to generate the MCLK to codec.

Remarks

None.

Building the Library

This section lists the files that are available in the AK4645 Codec Driver Library.

Description

This section list the files that are available in the `/src` folder of the AK4645 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/codec/ak4645`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_ak4645.h	Header file that exports the driver API.

Required File(s)

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_ak4645.c	This file contains implementation of the AK4645 Codec Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
There are no optional files for this driver.	N/A

Module Dependencies

The AK4645 Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_AK4645_Initialize	Initializes hardware and data for the instance of the AK4645 DAC module
	DRV_AK4645_Deinitialize	Deinitializes the specified instance of the AK4645 driver module
	DRV_AK4645_Status	Gets the current status of the AK4645 driver module.
	DRV_AK4645_Tasks	Maintains the driver's control and data interface state machine.

b) Client Setup Functions

	Name	Description
	DRV_AK4645_Open	Opens the specified AK4645 driver instance and returns a handle to it
	DRV_AK4645_Close	Closes an opened-instance of the AK4645 driver

c) Codec Specific Functions

	Name	Description
	DRV_AK4645_IntExtMicSet	This function sets up the codec for the internal or the external microphone use.
	DRV_AK4645_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4645_MuteOff	This function disables AK4645 output for soft mute.
	DRV_AK4645_MuteOn	This function allows AK4645 output for soft mute on.
	DRV_AK4645_SamplingRateGet	This function gets the sampling rate set on the AK4645. **Implementation:** Dynamic
	DRV_AK4645_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_AK4645_VolumeGet	This function gets the volume for AK4645 CODEC.
	DRV_AK4645_VolumeSet	This function sets the volume for AK4645 CODEC.

d) Data Transfer Functions

	Name	Description
	DRV_AK4645_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4645_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_AK4645_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4645_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

e) Other Functions

	Name	Description
	DRV_AK4645_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.
	DRV_AK4645_VersionGet	This function returns the version of AK4645 driver
	DRV_AK4645_VersionStrGet	This function returns the version of AK4645 driver in string format.

f) Data Types and Constants

	Name	Description
	DRV_AK4645_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4645_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4645_BUFFER_EVENT_HANDLER	Pointer to a AK4645 Driver Buffer Event handler function
	DRV_AK4645_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4645_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4645_COMMAND_EVENT_HANDLER	Pointer to a AK4645 Driver Command Event Handler Function
	DRV_AK4645_INIT	Defines the data required to initialize or reinitialize the AK4645 driver
	DRV_AK4645_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4645_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.
	DRV_AK4645_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4645_COUNT	Number of valid AK4645 driver indices
	DRV_AK4645_INDEX_0	AK4645 driver index definitions
	DRV_AK4645_INDEX_1	This is macro DRV_AK4645_INDEX_1.
	DRV_AK4645_INDEX_2	This is macro DRV_AK4645_INDEX_2.
	DRV_AK4645_INDEX_3	This is macro DRV_AK4645_INDEX_3.
	DRV_AK4645_INDEX_4	This is macro DRV_AK4645_INDEX_4.
	DRV_AK4645_INDEX_5	This is macro DRV_AK4645_INDEX_5.

Description

This section describes the API functions of the AK4645 Codec Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_AK4645_Initialize Function

Initializes hardware and data for the instance of the AK4645 DAC module

File

drv_ak4645.h

C

```
SYS_MODULE_OBJ DRV_AK4645_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the AK4645 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Remarks

This routine must be called before any other AK4645 routine is called.

This routine should only be called once during system initialization unless DRV_AK4645_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this CODEC driver. DRV_I2C_Initialize must be called if SPI driver is used for handling the control interface of this CODEC driver.

Example

```
DRV_AK4645_INIT                 init;
SYS_MODULE_OBJ                  objectHandle;

init->inUse                           = true;
init->status                          = SYS_STATUS_BUSY;
init->numClients                      = 0;
init->i2sDriverModuleIndex            = ak4645Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex            = ak4645Init->i2cDriverModuleIndex;
init->samplingRate                    = DRV_AK4645_AUDIO_SAMPLING_RATE;
init->audioDataFormat                 = DRV_AK4645_AUDIO_DATA_FORMAT_MACRO;

init->isInInterruptContext            = false;

init->commandCompleteCallback = (DRV_AK4645_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;
init->mclk_multiplier = DRV_AK4645_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_AK4645_Initialize(DRV_AK4645_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_AK4645_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT *const init

);

DRV_AK4645_Deinitialize Function

Deinitializes the specified instance of the AK4645 driver module

File

drv_ak4645.h

C

```
void DRV_AK4645_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the AK4645 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_AK4645_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_AK4645_Initialize
SYS_STATUS          status;


DRV_AK4645_Deinitialize(object);

status = DRV_AK4645_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4645_Initialize routine

Function

void DRV_AK4645_Deinitialize(SYS_MODULE_OBJ object)

DRV_AK4645_Status Function

Gets the current status of the AK4645 driver module.

File

drv_ak4645.h

C

```
SYS_STATUS DRV_AK4645_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This routine provides the current status of the AK4645 driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_AK4645_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4645_Initialize
SYS_STATUS          AK4645Status;

AK4645Status = DRV_AK4645_Status(object);
if (SYS_STATUS_READY == AK4645Status)
{
    // This means the driver can be opened using the
    // DRV_AK4645_Open( ) function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4645_Initialize routine

Function

SYS_STATUS DRV_AK4645_Status(SYS_MODULE_OBJ object)

DRV_AK4645_Tasks Function

Maintains the driver's control and data interface state machine.

File

drv_ak4645.h

C

```
void DRV_AK4645_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4645_Initialize

while (true)
{
    DRV_AK4645_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4645_Initialize)

Function

void DRV_AK4645_Tasks(SYS_MODULE_OBJ object);

b) Client Setup Functions

DRV_AK4645_Open Function

Opens the specified AK4645 driver instance and returns a handle to it

File

drv_ak4645.h

C

```
DRV_HANDLE DRV_AK4645_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_AK4645_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

Description

This routine opens the specified AK4645 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

AK4645 can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_AK4645_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

Function DRV_AK4645_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_AK4645_Open(DRV_AK4645_INDEX_0, DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_AK4645_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

)

DRV_AK4645_Close Function

Closes an opened-instance of the AK4645 driver

File

drv_ak4645.h

C

```
void DRV_AK4645_Close(const DRV_HANDLE handle);
```

Returns

- None

Description

This routine closes an opened-instance of the AK4645 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_AK4645_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 driver instance.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_AK4645_Open

DRV_AK4645_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4645_Close(DRV_Handle handle)

c) Codec Specific Functions

DRV_AK4645_IntExtMicSet Function

This function sets up the codec for the internal or the external microphone use.

File

drv_ak4645.h

C

```
void DRV_AK4645_IntExtMicSet(DRV_HANDLE handle, DRV_AK4645_INT_EXT_MIC micInput);
```

Returns

None

Description

This function sets up the codec for the internal or the external microphone use.

Remarks

None.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 driver instance.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	Internal vs External mic input

Function

void DRV_AK4645_IntExtMicSet(DRV_HANDLE handle);

DRV_AK4645_MonoStereoMicSet Function

This function sets up the codec for the Mono or Stereo microphone mode.

File

drv_ak4645.h

C

```
void DRV_AK4645_MonoStereoMicSet(DRV_HANDLE handle, DRV_AK4645_MONO_STEREO_MIC mono_stereo_mic);
```

Returns

None

Description

This function sets up the codec for the Mono or Stereo microphone mode.

Remarks

None.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 driver instance.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4645_MonoStereoMicSet(DRV_HANDLE handle);

DRV_AK4645_MuteOff Function

This function disables AK4645 output for soft mute.

File

drv_ak4645.h

C

```
void DRV_AK4645_MuteOff(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables AK4645 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 driver instance.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4645Handle is the handle returned
// by the DRV_AK4645_Open function.

    DRV_AK4645_MuteOff(myAK4645Handle); //AK4645 output soft mute disabled
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4645_MuteOff(DRV_HANDLE handle)

DRV_AK4645_MuteOn Function

This function allows AK4645 output for soft mute on.

File

drv_ak4645.h

C

```
void DRV_AK4645_MuteOn(DRV_HANDLE handle);
```

Returns

None.

Description

This function Enables AK4645 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 driver instance.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4645Handle is the handle returned
// by the DRV_AK4645_Open function.

DRV_AK4645_MuteOn(myAK4645Handle);  //AK4645 output soft muted
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4645_MuteOn(DRV_HANDLE handle);

DRV_AK4645_SamplingRateGet Function

This function gets the sampling rate set on the AK4645.

Implementation: Dynamic

File

drv_ak4645.h

C

```
uint32_t DRV_AK4645_SamplingRateGet(DRV_HANDLE handle);
```

Description

This function gets the sampling rate set on the DAC AK4645.

Remarks

None.

Example

```
uint32_t baudRate;

// myAK4645Handle is the handle returned
// by the DRV_AK4645_Open function.

baudRate = DRV_AK4645_SamplingRateGet(myAK4645Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_AK4645_SamplingRateGet(DRV_HANDLE handle)

DRV_AK4645_SamplingRateSet Function

This function sets the sampling rate of the media stream.

File

drv_ak4645.h

C

```
void DRV_AK4645_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

Returns

None.

Description

This function sets the media sampling rate for the client handle.

Remarks

None.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 driver instance.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

Example

```
// myAK4645Handle is the handle returned
// by the DRV_AK4645_Open function.

DRV_AK4645_SamplingRateSet(myAK4645Handle, 48000);  //Sets 48000 media sampling rate
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
samplingRate	Sampling frequency in Hz

Function

void DRV_AK4645_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

DRV_AK4645_VolumeGet Function

This function gets the volume for AK4645 CODEC.

File

drv_ak4645.h

C

```
uint8_t DRV_AK4645_VolumeGet(DRV_HANDLE handle, DRV_AK4645_CHANNEL channel);
```

Returns

None.

Description

This functions gets the current volume programmed to the CODEC AK4645.

Remarks

None.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 driver instance.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK4645Handle is the handle returned
// by the DRV_AK4645_Open function.

  volume = DRV_AK4645_VolumeGet(myAK4645Handle, DRV_AK4645_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified

Function

uint8_t DRV_AK4645_VolumeGet(DRV_HANDLE handle, DRV_AK4645_CHANNEL channel)

DRV_AK4645_VolumeSet Function

This function sets the volume for AK4645 CODEC.

File

drv_ak4645.h

C

```
void DRV_AK4645_VolumeSet(DRV_HANDLE handle, DRV_AK4645_CHANNEL channel, uint8_t volume);
```

Returns

None

Description

This functions sets the volume value from 0-255. The codec has DAC value to volume range mapping as :- 00 H : +12dB FF H : -115dB In order to make the volume value to dB mapping monotonically increasing from 00 to FF, re-mapping is introduced which reverses the volume value to dB mapping as well as normalizes the volume range to a more audible dB range. The current driver implementation assumes that all dB values under -60 dB are inaudible to the human ear. Re-Mapped values 00 H : -60 dB FF H : +12 dB

Remarks

None.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 driver instance.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4645Handle is the handle returned
// by the DRV_AK4645_Open function.

    DRV_AK4645_VolumeSet(myAK4645Handle,DRV_AK4645_CHANNEL_LEFT, 120);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified
volume	volume value specified in the range 0-255 (0x00 to 0xFF)

Function

void DRV_AK4645_VolumeSet(DRV_HANDLE handle, DRV_AK4645_CHANNEL channel, uint8_t volume);

d) Data Transfer Functions

DRV_AK4645_BufferAddRead Function

Schedule a non-blocking driver read operation.

File

drv_ak4645.h

C

```
void DRV_AK4645_BufferAddRead(const DRV_HANDLE handle, DRV_AK4645_BUFFER_HANDLE * bufferHandle,
void * buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4645_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4645_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4645_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4645_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4645 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4645 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 device instance and the DRV_AK4645_Status must have returned SYS_STATUS_READY.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_AK4645_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4645 instance as return by the DRV_AK4645_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4645_BufferAddRead

(
const DRV_HANDLE handle,
DRV_AK4645_BUFFER_HANDLE *bufferHandle,
void *buffer, size_t size
)

DRV_AK4645_BufferAddWrite Function

Schedule a non-blocking driver write operation.

File

drv_ak4645.h

C

```
void DRV_AK4645_BufferAddWrite(const DRV_HANDLE handle, DRV_AK4645_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4645_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4645_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4645_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4645_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4645 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4645 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 device instance and the DRV_AK4645_Status must have returned SYS_STATUS_READY.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK4645_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4645_BUFFER_HANDLE bufferHandle;

// myAK4645Handle is the handle returned
// by the DRV_AK4645_Open function.

// Client registers an event handler with driver
```

```
DRV_AK4645_BufferEventHandlerSet(myAK4645Handle,
                APP_AK4645BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4645_BufferAddWrite(myAK4645handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4645_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4645BufferEventHandler(DRV_AK4645_BUFFER_EVENT event,
        DRV_AK4645_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4645_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4645_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4645 instance as return by the DRV_AK4645_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4645_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_AK4645_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

DRV_AK4645_BufferAddWriteRead Function

Schedule a non-blocking driver write-read operation.

Implementation: Dynamic

File

drv_ak4645.h

C

```
void DRV_AK4645_BufferAddWriteRead(const DRV_HANDLE handle, DRV_AK4645_BUFFER_HANDLE *
bufferHandle, void * transmitBuffer, void * receiveBuffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4645_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4645_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4645_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4645_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4645 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4645 driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every AK4645 write. The transmit and receive size must be same.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 device instance and the DRV_AK4645_Status must have returned SYS_STATUS_READY.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_AK4645_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_AK4645_BUFFER_HANDLE bufferHandle;

// myak4645Handle is the handle returned
// by the DRV_AK4645_Open function.

// Client registers an event handler with driver

DRV_AK4645_BufferEventHandlerSet(myak4645Handle,
                APP_AK4645BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4645_BufferAddWriteRead(myak4645handle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_AK4645_BUFFER_HANDLE_INVALID == bufferHandle)
{
```

```
        // Error handling here
    }

    // Event is received when
    // the buffer is processed.

    void APP_AK4645BufferEventHandler(DRV_AK4645_BUFFER_EVENT event,
            DRV_AK4645_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
    {
        // contextHandle points to myAppObj.

        switch(event)
        {
            case DRV_AK4645_BUFFER_EVENT_COMPLETE:

                // This means the data was transferred.
                break;

            case DRV_AK4645_BUFFER_EVENT_ERROR:

                // Error handling here.
                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	Handle of the AK4645 instance as returned by the DRV_AK4645_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	The buffer where the transmit data will be stored
receiveBuffer	The buffer where the received data will be stored
size	Buffer size in bytes

Function

void DRV_AK4645_BufferAddWriteRead

(

const DRV_HANDLE handle,

DRV_AK4645_BUFFER_HANDLE *bufferHandle,

void *transmitBuffer,

void *receiveBuffer,

size_t size

)

DRV_AK4645_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

File

drv_ak4645.h

C

```
void DRV_AK4645_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK4645_BUFFER_EVENT_HANDLER
```

```
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_AK4645_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 driver instance.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4645_BUFFER_HANDLE bufferHandle;

// myAK4645Handle is the handle returned
// by the DRV_AK4645_Open function.

// Client registers an event handler with driver

DRV_AK4645_BufferWriteEventHandlerSet(myAK4645Handle,
                APP_AK4645BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4645_BufferAddWrite(myAK4645handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4645_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4645BufferEventHandler(DRV_AK4645_BUFFER_EVENT event,
        DRV_AK4645_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4645_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4645_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;
```

```
        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4645_BufferWriteEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4645_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

e) Other Functions

DRV_AK4645_CommandEventHandlerSet Function

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

File

drv_ak4645.h

C

```
void DRV_AK4645_CommandEventHandlerSet(DRV_HANDLE handle, const
DRV_AK4645_COMMAND_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK4645_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "AK4645 CODEC Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Preconditions

The DRV_AK4645_Initialize routine must have been called for the specified AK4645 driver instance.

DRV_AK4645_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4645_BUFFER_HANDLE bufferHandle;

// myAK4645Handle is the handle returned
// by the DRV_AK4645_Open function.

// Client registers an event handler with driver

DRV_AK4645_CommandEventHandlerSet(myAK4645Handle,
                APP_AK4645CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK4645_DeEmphasisFilterSet(myAK4645Handle, DRV_AK4645_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK4645CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
```

```
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4645_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4645_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4645_VersionGet Function

This function returns the version of AK4645 driver

File

drv_ak4645.h

C

```
uint32_t DRV_AK4645_VersionGet();
```

Returns

returns the version of AK4645 driver.

Description

The version number returned from the DRV_AK4645_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Remarks

None.

Preconditions

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
uint32_t AK4645version;
AK4645version = DRV_AK4645_VersionGet();
```

Function

uint32_t DRV_AK4645_VersionGet(void)

DRV_AK4645_VersionStrGet Function

This function returns the version of AK4645 driver in string format.

File

drv_ak4645.h

C

```
int8_t* DRV_AK4645_VersionStrGet();
```

Returns

returns a string containing the version of AK4645 driver.

Description

The DRV_AK4645_VersionStrGet function returns a string in the format: ".[.][]" Where: is the AK4645 driver's version number. is the AK4645 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces. For example, "0.03a" "1.00"

Remarks

None

Preconditions

None.

Example

```
int8_t *AK4645string;
AK4645string = DRV_AK4645_VersionStrGet();
```

Function

int8_t* DRV_AK4645_VersionStrGet(void)

f) Data Types and Constants

DRV_AK4645_AUDIO_DATA_FORMAT Enumeration

Identifies the Serial Audio data interface format.

File

drv_ak4645.h

C

```
typedef enum {
  DRV_AK4645_AUDIO_DATA_FORMAT_DSP = 0,
  DRV_AK4645_AUDIO_DATA_FORMAT_16BITMSB_SDTO_16BITLSB_SDTI,
  DRV_AK4645_AUDIO_DATA_FORMAT_16BITMSB_SDTO_16BITMSB_SDTI,
  DRV_AK4645_AUDIO_DATA_FORMAT_I2S
} DRV_AK4645_AUDIO_DATA_FORMAT;
```

Description

AK4645 Audio data format

This enumeration identifies Serial Audio data interface format.

DRV_AK4645_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_ak4645.h

C

```
typedef enum {
  DRV_AK4645_BUFFER_EVENT_COMPLETE,
  DRV_AK4645_BUFFER_EVENT_ERROR,
  DRV_AK4645_BUFFER_EVENT_ABORT
} DRV_AK4645_BUFFER_EVENT;
```

Members

Members	Description
DRV_AK4645_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_AK4645_BUFFER_EVENT_ERROR	Error while processing the request
DRV_AK4645_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

AK4645 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_AK4645_BufferAddWrite() or the DRV_AK4645_BufferAddRead() function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_AK4645_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_AK4645_BUFFER_EVENT_HANDLER Type

Pointer to a AK4645 Driver Buffer Event handler function

File

drv_ak4645.h

C

```
typedef void (* DRV_AK4645_BUFFER_EVENT_HANDLER)(DRV_AK4645_BUFFER_EVENT event,
DRV_AK4645_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

AK4645 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK4645 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_AK4645_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK4645_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_AK4645_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4645_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver(i2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_AK4645_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_AK4645_BUFFER_EVENT event,
                               DRV_AK4645_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK4645_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK4645_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
```

```
            break;
        }
    }
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4645_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

File

drv_ak4645.h

C

```
typedef uintptr_t DRV_AK4645_BUFFER_HANDLE;
```

Description

AK4645 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK4645_BufferAddWrite() or DRV_AK4645_BufferAddRead() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_AK4645_CHANNEL Enumeration

Identifies Left/Right Audio channel

File

drv_ak4645.h

C

```
typedef enum {
  DRV_AK4645_CHANNEL_LEFT,
  DRV_AK4645_CHANNEL_RIGHT,
  DRV_AK4645_CHANNEL_LEFT_RIGHT,
  DRV_AK4645_NUMBER_OF_CHANNELS
} DRV_AK4645_CHANNEL;
```

Description

AK4645 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

DRV_AK4645_COMMAND_EVENT_HANDLER Type

Pointer to a AK4645 Driver Command Event Handler Function

File

drv_ak4645.h

C

```
typedef void (* DRV_AK4645_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

Returns

None.

Description

AK4645 Driver Command Event Handler Function

This data type defines the required function signature for the AK4645 driver command event handling callback function.

A command is a control instruction to the AK4645 CODEC. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4645_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_AK4645CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4645_INIT Structure

Defines the data required to initialize or reinitialize the AK4645 driver

File

drv_ak4645.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX i2sDriverModuleIndex;
  SYS_MODULE_INDEX i2cDriverModuleIndex;
  uint32_t samplingRate;
  uint8_t volume;
  DRV_AK4645_AUDIO_DATA_FORMAT audioDataFormat;
} DRV_AK4645_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX i2sDriverModuleIndex;	Identifies data module(I2S) driver ID for data interface of CODEC
SYS_MODULE_INDEX i2cDriverModuleIndex;	Identifies data module(I2C) driver ID for control interface of CODEC
uint32_t samplingRate;	Sampling rate
uint8_t volume;	Volume
DRV_AK4645_AUDIO_DATA_FORMAT audioDataFormat;	Identifies the Audio data format

Description

AK4645 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK4645 CODEC driver.

Remarks

None.

DRV_AK4645_INT_EXT_MIC Enumeration

Identifies the Mic input source.

File

drv_ak4645.h

C

```
typedef enum {
  INT_MIC,
  EXT_MIC
} DRV_AK4645_INT_EXT_MIC;
```

Description

AK4645 Mic Internal / External Input

This enumeration identifies the Mic input source.

DRV_AK4645_MONO_STEREO_MIC Enumeration

Identifies the Mic input as Mono / Stereo.

File

drv_ak4645.h

C

```
typedef enum {
  ALL_ZEROS,
  MONO_RIGHT_CHANNEL,
  MONO_LEFT_CHANNEL,
  STEREO
} DRV_AK4645_MONO_STEREO_MIC;
```

Description

AK4645 Mic Mono / Stereo Input

This enumeration identifies the Mic input as Mono / Stereo.

DRV_AK4645_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_ak4645.h

C

```
#define DRV_AK4645_BUFFER_HANDLE_INVALID ((DRV_AK4645_BUFFER_HANDLE)(-1))
```

Description

AK4645 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_AK4645_BufferAddWrite() and the DRV_AK4645_BufferAddRead() function if the buffer add request was not successful.

Remarks

None.

DRV_AK4645_COUNT Macro

Number of valid AK4645 driver indices

File

drv_ak4645.h

C

```
#define DRV_AK4645_COUNT
```

Description

AK4645 Driver Module Count

This constant identifies the maximum number of AK4645 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of AK4645 instances on this microcontroller.

Remarks

This value is part-specific.

DRV_AK4645_INDEX_0 Macro

AK4645 driver index definitions

File

drv_ak4645.h

C

```
#define DRV_AK4645_INDEX_0 0
```

Description

Driver AK4645 Module Index

These constants provide AK4645 driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_AK4645_Initialize and DRV_AK4645_Open routines to identify the driver instance in use.

DRV_AK4645_INDEX_1 Macro

File

drv_ak4645.h

C

```
#define DRV_AK4645_INDEX_1 1
```

Description

This is macro DRV_AK4645_INDEX_1.

DRV_AK4645_INDEX_2 Macro

File

drv_ak4645.h

C

```
#define DRV_AK4645_INDEX_2 2
```

Description

This is macro DRV_AK4645_INDEX_2.

DRV_AK4645_INDEX_3 Macro

File

drv_ak4645.h

C

```
#define DRV_AK4645_INDEX_3 3
```

Description

This is macro DRV_AK4645_INDEX_3.

DRV_AK4645_INDEX_4 Macro

File

drv_ak4645.h

C

```
#define DRV_AK4645_INDEX_4 4
```

Description

This is macro DRV_AK4645_INDEX_4.

DRV_AK4645_INDEX_5 Macro

File

drv_ak4645.h

C

```
#define DRV_AK4645_INDEX_5 5
```

Description

This is macro DRV_AK4645_INDEX_5.

Files

Files

Name	Description
drv_ak4645.h	AK4645 CODEC Driver Interface header file
drv_ak4645_config_template.h	AK4645 Codec Driver Configuration Template.

Description

This section lists the source and header files used by the AK4645 Codec Driver Library.

drv_ak4645.h

AK4645 CODEC Driver Interface header file

Enumerations

	Name	Description
	DRV_AK4645_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4645_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4645_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4645_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4645_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.

Functions

	Name	Description
	DRV_AK4645_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4645_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_AK4645_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4645_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_AK4645_Close	Closes an opened-instance of the AK4645 driver
	DRV_AK4645_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.
	DRV_AK4645_Deinitialize	Deinitializes the specified instance of the AK4645 driver module
	DRV_AK4645_Initialize	Initializes hardware and data for the instance of the AK4645 DAC module
	DRV_AK4645_IntExtMicSet	This function sets up the codec for the internal or the external microphone use.
	DRV_AK4645_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4645_MuteOff	This function disables AK4645 output for soft mute.
	DRV_AK4645_MuteOn	This function allows AK4645 output for soft mute on.
	DRV_AK4645_Open	Opens the specified AK4645 driver instance and returns a handle to it
	DRV_AK4645_SamplingRateGet	This function gets the sampling rate set on the AK4645. **Implementation:** Dynamic
	DRV_AK4645_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_AK4645_Status	Gets the current status of the AK4645 driver module.
	DRV_AK4645_Tasks	Maintains the driver's control and data interface state machine.
	DRV_AK4645_VersionGet	This function returns the version of AK4645 driver
	DRV_AK4645_VersionStrGet	This function returns the version of AK4645 driver in string format.
	DRV_AK4645_VolumeGet	This function gets the volume for AK4645 CODEC.
	DRV_AK4645_VolumeSet	This function sets the volume for AK4645 CODEC.

Macros

	Name	Description
	DRV_AK4645_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4645_COUNT	Number of valid AK4645 driver indices
	DRV_AK4645_INDEX_0	AK4645 driver index definitions
	DRV_AK4645_INDEX_1	This is macro DRV_AK4645_INDEX_1.
	DRV_AK4645_INDEX_2	This is macro DRV_AK4645_INDEX_2.
	DRV_AK4645_INDEX_3	This is macro DRV_AK4645_INDEX_3.
	DRV_AK4645_INDEX_4	This is macro DRV_AK4645_INDEX_4.
	DRV_AK4645_INDEX_5	This is macro DRV_AK4645_INDEX_5.

Structures

	Name	Description
	DRV_AK4645_INIT	Defines the data required to initialize or reinitialize the AK4645 driver

Types

	Name	Description
	DRV_AK4645_BUFFER_EVENT_HANDLER	Pointer to a AK4645 Driver Buffer Event handler function
	DRV_AK4645_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4645_COMMAND_EVENT_HANDLER	Pointer to a AK4645 Driver Command Event Handler Function

Description

AK4645 CODEC Driver Interface

The AK4645 CODEC device driver interface provides a simple interface to manage the AK4645 16/24-Bit CODEC that can be interfaced Microchip Microcontroller. This file provides the interface definition for the AK4645 CODEC device driver.

File Name

drv_ak4645.h

Company

Microchip Technology Inc.

drv_ak4645_config_template.h

AK4645 Codec Driver Configuration Template.

Macros

	Name	Description
	DRV_AK4645_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4645_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4645_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4645_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4645_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4645_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.

Description

AK4645 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_ak4645_config_template.h

Company

Microchip Technology Inc.

AK4953 Codec Driver Library

This topic describes the AK4953 Codec Driver Library.

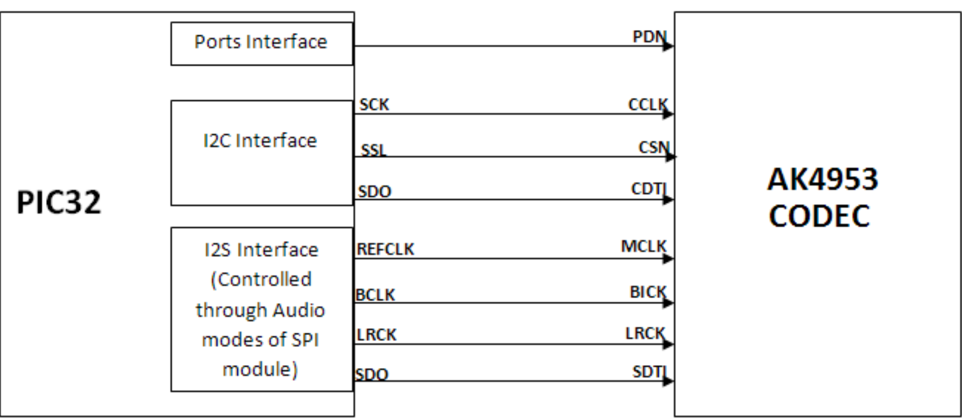
Introduction

This library provides an interface to manage the AK4953 Codec that is serially interfaced to a Microchip microcontroller for providing Audio Solutions.

Description

The AK4953 module is 16/24-bit Audio Codec from Asahi Kasei Microdevices Corporation. The AK4953 can be interfaced to Microchip microcontrollers through I2C and I2S serial interfaces. The I2C interface is used for control command transfer. The I2S interface is used for Audio data output.

A typical interface of AK4953 to a Microchip PIC32 device is provided in the following diagram:



Features

The AK4953 Codec supports the following features:

- Audio Interface Format: MSB first
- ADC: 24-bit MSB justified, 16/24-bit I2S
- DAC: 24-bit MSB justified, 1-6bit LSB justified, 24-bit LSB justified, 16/24-bit I2S
- Sampling Frequency Range: 8 kHz to 192 kHz
- Digital Volume Control: +12dB ~ .115dB, 0.5dB Step
- SoftMute: On and Off
- Master Clock Frequencies: 32 fs/64 fs/128 fs/256 fs

Using the Library

This topic describes the basic architecture of the AK4953 Codec Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_AK4953.h`

The interface to the AK4953 Codec Driver library is defined in the `drv_AK4953.h` header file. Any C language source (`.c`) file that uses the AK4953 Codec Driver library should include this header.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The AK4953 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK4953 DAC module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK4953 Codec Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Status Functions	Provides status functions.
Other Functions	Provides driver specific miscellaneous functions such as sampling rate setting, control command functions, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the AK4953 Codec Driver Library.

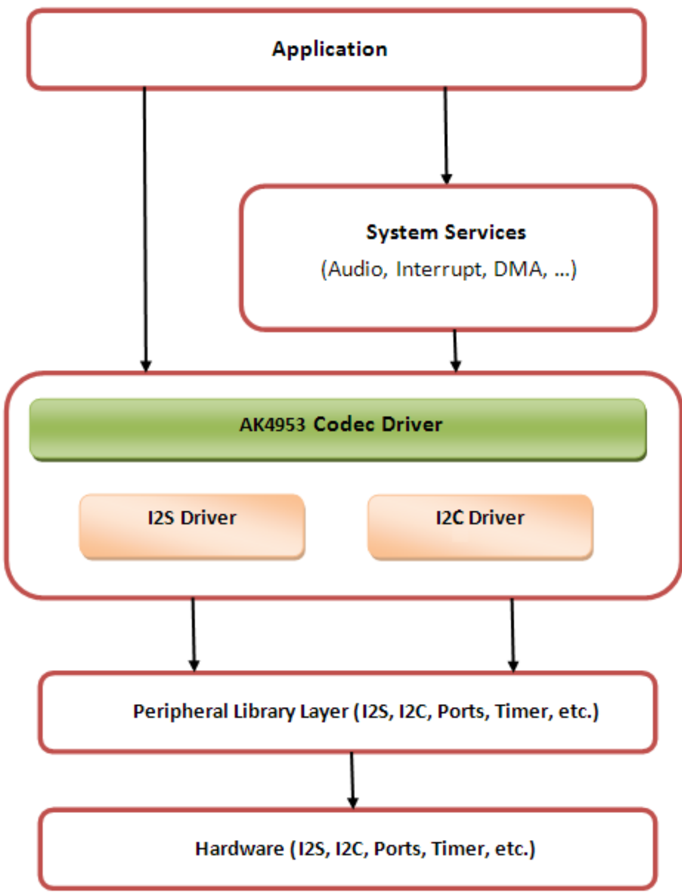
Abstraction Model

This library provides a low-level abstraction of the AK4953 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the AK4953 Codec Driver is positioned in the MPLAB Harmony framework. The AK4953 Codec Driver uses the SPI and I2S drivers for control and audio data transfers to the AK4953 module.

AK4953 Driver Abstraction Model



How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

System Access

This topic describes system initialization, implementations, and includes a system access code example.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the AK4953 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK4953_INIT or by using Initialization Overrides) that are supported by the specific AK4953 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver.
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver.
- Sampling rate
- Audio data format. The audio data format should match with the audio data format settings done in I2S driver initialization
- Power down pin port initialization
- Queue size for the audio data transmit buffer

The DRV_AK4953_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ AK4953_Deinitialize, DRV_ AK4953_Status and DRV_I2S_Tasks.

Implementations

The AK4953 Codec Driver can has the following implementation:

Description	MPLAB Harmony Components
Dedicated hardware for control (I2C) and data (I2S) interface.	Standard MPLAB Harmony drivers for I2C and I2S interfaces.

Example:

```
DRV_AK4953_INIT drvak4953Codec0InitData =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .i2sDriverModuleIndex = DRV_AK4953_I2S_DRIVER_MODULE_INDEX_IDX0,
    .i2cDriverModuleIndex = DRV_AK4953_I2C_DRIVER_MODULE_INDEX_IDX0,
    .volume = DRV_AK4953_VOLUME,
    .queueSizeTransmit = DRV_AK4953_TRANSMIT_QUEUE_SIZE,
};

// Initialize the I2C driver
   DRV_I2C0_Initialize();

// Initialize the I2S driver. The I2S module index should be same as the one used in initializing
// the I2S driver.
   sysObj.drvI2S0 = DRV_I2S_Initialize(DRV_I2S_INDEX_0, (SYS_MODULE_INIT *)&drvI2S0InitData);

// Initialize the Codec driver
```

```
   sysObj.drvak4953Codec0 = DRV_AK4953_Initialize(DRV_AK4953_INDEX_0, (SYS_MODULE_INIT
*)&drvak4953Codec0InitData);

if (SYS_MODULE_OBJ_INVALID == AK4953DevObject)
{
// Handle error
}
```

Task Routine

The DRV_AK4953_Tasks will be called from the System Task Service.

Client Access

For the application to start using an instance of the module, it must call the DRV_AK4953_Open function. The DRV_AK4953_Open provides a driver handle to the AK4953 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK4953_Deinitialize, the application must call the DRV_AK4953_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

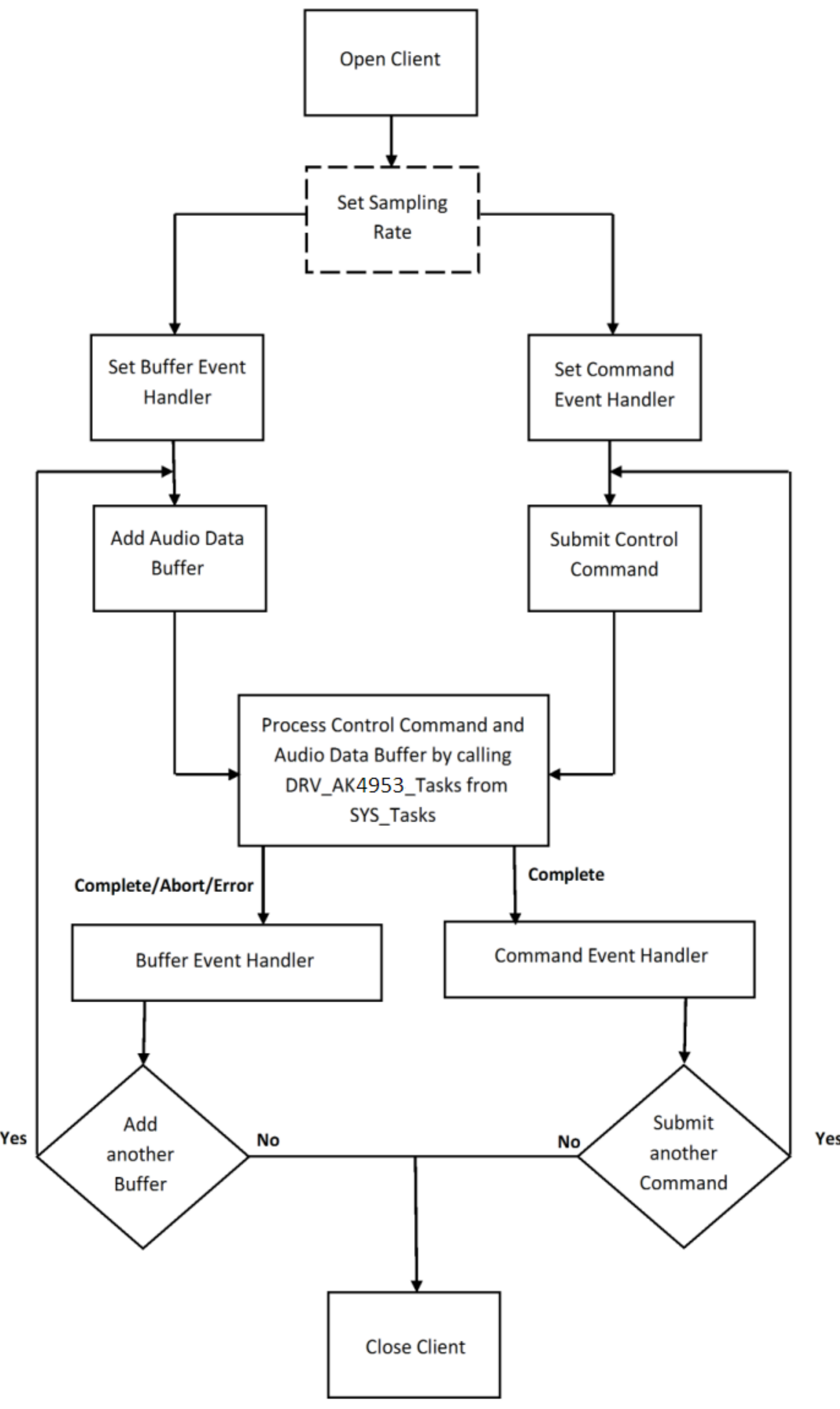
Client Operations

This topic provides information on client operations and includes a control command and audio buffered data operation flow diagram.

Description

Client operations provide the API interface for control command and audio data transfer to the AK4953 DAC.

The following diagram illustrates the control commands and audio buffered data operations.



Note: It is not necessary to close and reopen the client between multiple transfers.

Configuring the Library

Macros

	Name	Description
	DRV_AK4953_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4953_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4953_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4953_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4953_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.
	DRV_AK4953_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.

Description

The configuration of the AK4953 Codec Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the AK4953 Codec Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the AK4953 Codec Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

DRV_AK4953_BCLK_BIT_CLK_DIVISOR Macro

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_BCLK_BIT_CLK_DIVISOR
```

Description

AK4953 BCLK to LRCK Ratio to Generate Audio Stream

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

Following BCLK to LRCK ratios are supported 16bit data 16 bit channel :- 32fs, hence divisor would be 8 16bit data 32 bit channel :- 64fs, hence divisor would be 4

Remarks

None.

DRV_AK4953_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_CLIENTS_NUMBER DRV_AK4953_INSTANCES_NUMBER
```

Description

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances. Therefore, if there are five AK4953 hardware interfaces, this number will be 5.

Remarks

None.

Section

Client Configuration

****

****

****

AK4953 Client Count Configuration

DRV_AK4953_INPUT_REFCLOCK Macro

Identifies the input REFCLOCK source to generate the MCLK to codec.

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_INPUT_REFCLOCK
```

Description

AK4953 Input reference clock

Identifies the input REFCLOCK source to generate the MCLK to codec.

Remarks

None.

DRV_AK4953_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_INSTANCES_NUMBER
```

Description

AK4953 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of AK4953 CODEC modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER Macro

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER
```

Description

AK4953 MCLK to LRCK Ratio to Generate Audio Stream

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency I2S sampling frequency

Supported MCLK to Sampling frequency Ratios are as below 256fs, 384fs, 512fs, 768fs or 1152fs

Remarks

None

DRV_AK4953_MCLK_SOURCE Macro

Indicate the input clock frequency to generate the MCLK to codec.

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_MCLK_SOURCE
```

Description

Indicate the input clock frequency to generate the MCLK to codec.

Remarks

None.

Section

CODEC Specific Configuration

```
********************************************************************************
********************************************************************************
********************************************************************************
********************************************************************************
```

AK4953 Data Interface Master Clock Speed configuration

DRV_AK4953_QUEUE_DEPTH_COMBINED Macro

Number of entries of all queues in all instances of the driver.

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_QUEUE_DEPTH_COMBINED
```

Description

AK4953 Driver Buffer Queue Entries

This macro defined the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for transmit operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build). The hardware instance transmit buffer queue will queue transmit buffers submitted by the DRV_AK4953_BufferAddWrite function.

A buffer queue will contains buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all AK4953 driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking write requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its transmit buffer queue size.

As an example, consider the case of static single client driver application where full duplex non blocking operation is desired without queuing, the minimum transmit queue depth and minimum receive queue depth should be 1. Hence the total number of buffer entries should be 2.

As an example, consider the case of a dynamic driver (say two instances) where instance one will queue up to three write requests and up to two read requests, and instance two will queue up to two write requests and up to six read requests, the value of this macro should be 13 (2 + 3 + 2 + 6).

Building the Library

This section lists the files that are available in the AK4953 Codec Driver Library.

Description

This section list the files that are available in the `/src` folder of the AK4953 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/codec/ak4953`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_ak4953.h	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_ak4953.c	This file contains implementation of the AK4953 Codec Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The AK4953 Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_AK4953_Initialize	Initializes hardware and data for the instance of the AK4953 DAC module. **Implementation:** Dynamic
	DRV_AK4953_Deinitialize	Deinitializes the specified instance of the AK4953 driver module. **Implementation:** Dynamic
	DRV_AK4953_Open	Opens the specified AK4953 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_AK4953_Close	Closes an opened-instance of the AK4953 driver. **Implementation:** Dynamic
	DRV_AK4953_Tasks	Maintains the driver's control and data interface state machine. **Implementation:** Dynamic
	DRV_AK4953_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. **Implementation:** Dynamic
	DRV_AK4953_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_AK4953_SamplingRateSet	This function sets the sampling rate of the media stream. **Implementation:** Dynamic

b) Status Functions

	Name	Description
	DRV_AK4953_SamplingRateGet	This function gets the sampling rate set on the DAC AK4953. **Implementation:** Dynamic

	Name	Description
	DRV_AK4953_Status	Gets the current status of the AK4953 driver module. **Implementation:** Dynamic
	DRV_AK4953_VersionGet	This function returns the version of AK4953 driver. **Implementation:** Dynamic
	DRV_AK4953_VersionStrGet	This function returns the version of AK4953 driver in string format. **Implementation:** Dynamic
	DRV_AK4953_VolumeGet	This function gets the volume for AK4953 CODEC. **Implementation:** Dynamic

c) Other Functions

	Name	Description
	DRV_AK4953_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_AK4953_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4953_MuteOff	This function disables AK4953 output for soft mute. **Implementation:** Dynamic
	DRV_AK4953_MuteOn	This function allows AK4953 output for soft mute on. **Implementation:** Dynamic
	DRV_AK4953_VolumeSet	This function sets the volume for AK4953 CODEC. **Implementation:** Dynamic
	DRV_AK4953_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4953_IntExtMicSet	This function sets up the codec for the internal or the external microphone use.
	DRV_AK4953_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.

d) Data Types and Constants

	Name	Description
	DRV_AK4953_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4953_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4953_BUFFER_EVENT_HANDLER	Pointer to a AK4953 Driver Buffer Event handler function
	DRV_AK4953_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4953_COMMAND_EVENT_HANDLER	Pointer to a AK4953 Driver Command Event Handler Function
	DRV_AK4953_DIGITAL_BLOCK_CONTROL	Identifies Bass-Boost Control function
	DRV_AK4953_INIT	Defines the data required to initialize or reinitialize the AK4953 driver
	_DRV_AK4953_H	Include files.
	DRV_AK4953_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4953_COUNT	Number of valid AK4953 driver indices
	DRV_AK4953_INDEX_0	AK4953 driver index definitions
	DRV_AK4953_INDEX_1	This is macro DRV_AK4953_INDEX_1.
	DRV_AK4953_INDEX_2	This is macro DRV_AK4953_INDEX_2.
	DRV_AK4953_INDEX_3	This is macro DRV_AK4953_INDEX_3.
	DRV_AK4953_INDEX_4	This is macro DRV_AK4953_INDEX_4.
	DRV_AK4953_INDEX_5	This is macro DRV_AK4953_INDEX_5.

	DRV_I2C_INDEX	This is macro DRV_I2C_INDEX.
	DRV_AK4953_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4953_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4953_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.

Description

This section describes the API functions of the AK4953 Codec Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_AK4953_Initialize Function

Initializes hardware and data for the instance of the AK4953 DAC module.

Implementation: Dynamic

File

drv_ak4953.h

C

```
SYS_MODULE_OBJ DRV_AK4953_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the AK4953 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Remarks

This routine must be called before any other AK4953 routine is called.

This routine should only be called once during system initialization unless DRV_AK4953_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this CODEC driver. Also DRV_I2C_Initialize must be called before calling this function to initialize the control interface of this CODEC driver.

Example

```
DRV_AK4953_INIT                  init;
SYS_MODULE_OBJ                   objectHandle;

init->inUse                            = true;
init->status                           = SYS_STATUS_BUSY;
init->numClients                       = 0;
init->i2sDriverModuleIndex             = ak4953Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex             = ak4953Init->i2cDriverModuleIndex;
init->samplingRate                     = DRV_AK4953_AUDIO_SAMPLING_RATE;
init->audioDataFormat                  = DRV_AK4953_AUDIO_DATA_FORMAT_MACRO;
for(index=0; index < DRV_AK4953_NUMBER_OF_CHANNELS; index++)
{
    init->volume[index] = ak4953Init->volume;
}
init->isInInterruptContext             = false;

init->commandCompleteCallback = (DRV_AK4953_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;

init->mclk_multiplier = DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_AK4953_Initialize(DRV_AK4953_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
```

```
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_AK4953_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT *const init

);

DRV_AK4953_Deinitialize Function

Deinitializes the specified instance of the AK4953 driver module.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the AK4953 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_AK4953_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_AK4953_Initialize
SYS_STATUS          status;


DRV_AK4953_Deinitialize(object);

status = DRV_AK4953_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4953_Initialize routine

Function

void DRV_AK4953_Deinitialize(SYS_MODULE_OBJ object)

DRV_AK4953_Open Function

Opens the specified AK4953 driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_ak4953.h

C

```
DRV_HANDLE DRV_AK4953_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_AK4953_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

Description

This routine opens the specified AK4953 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

AK4953 can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_AK4953_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

Function DRV_AK4953_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_AK4953_Open(DRV_AK4953_INDEX_0, DRV_IO_INTENT_WRITEREAD | DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
```

```
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_AK4953_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

)

DRV_AK4953_Close Function

Closes an opened-instance of the AK4953 driver.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This routine closes an opened-instance of the AK4953 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_AK4953_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_AK4953_Open

DRV_AK4953_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4953_Close(DRV_Handle handle)

DRV_AK4953_Tasks Function

Maintains the driver's control and data interface state machine.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4953_Initialize

while (true)
{
    DRV_AK4953_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4953_Initialize)

Function

void DRV_AK4953_Tasks(SYS_MODULE_OBJ object);

DRV_AK4953_CommandEventHandlerSet Function

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_CommandEventHandlerSet(DRV_HANDLE handle, const
DRV_AK4953_COMMAND_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK4953_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "AK4953 CODEC Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4953_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_CommandEventHandlerSet(myAK4953Handle,
                APP_AK4953CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK4953_DeEmphasisFilterSet(myAK4953Handle, DRV_AK4953_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK4953CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4953_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4953_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4953_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

File

drv_ak4953.h

C

```
void DRV_AK4953_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK4953_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_AK4953_BufferAddRead function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
```

```
DRV_AK4953_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_BufferEventHandlerSet(myAK4953Handle,
                APP_AK4953BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4953_BufferAddRead(myAK4953handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4953_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4953BufferEventHandler(DRV_AK4953_BUFFER_EVENT event,
        DRV_AK4953_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4953_BufferEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4953_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4953_SamplingRateSet Function

This function sets the sampling rate of the media stream.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

Returns

None.

Description

This function sets the media sampling rate for the client handle.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

DRV_AK4953_SamplingRateSet(myAK4953Handle, 48000);  //Sets 48000 media sampling rate
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4953_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

b) Status Functions

DRV_AK4953_SamplingRateGet Function

This function gets the sampling rate set on the DAC AK4953.

Implementation: Dynamic

File

drv_ak4953.h

C

```
uint32_t DRV_AK4953_SamplingRateGet(DRV_HANDLE handle);
```

Returns

None.

Description

This function gets the sampling rate set on the DAC AK4953.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
uint32_t baudRate;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

baudRate = DRV_AK4953_SamplingRateGet(myAK4953Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_AK4953_SamplingRateGet(DRV_HANDLE handle)

DRV_AK4953_Status Function

Gets the current status of the AK4953 driver module.

Implementation: Dynamic

File

drv_ak4953.h

C

```
SYS_STATUS DRV_AK4953_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This routine provides the current status of the AK4953 driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_AK4953_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4953_Initialize
SYS_STATUS          AK4953Status;

AK4953Status = DRV_AK4953_Status(object);
if (SYS_STATUS_READY == AK4953Status)
{
    // This means the driver can be opened using the
    // DRV_AK4953_Open( ) function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4953_Initialize routine

Function

SYS_STATUS DRV_AK4953_Status(SYS_MODULE_OBJ object)

DRV_AK4953_VersionGet Function

This function returns the version of AK4953 driver.

Implementation: Dynamic

File

drv_ak4953.h

C

```
uint32_t DRV_AK4953_VersionGet();
```

Returns

returns the version of AK4953 driver.

Description

The version number returned from the DRV_AK4953_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Remarks

None.

Preconditions

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
uint32_t AK4953version;
AK4953version = DRV_AK4953_VersionGet();
```

Function

uint32_t DRV_AK4953_VersionGet(void)

DRV_AK4953_VersionStrGet Function

This function returns the version of AK4953 driver in string format.

Implementation: Dynamic

File

drv_ak4953.h

C

```
int8_t* DRV_AK4953_VersionStrGet();
```

Returns

returns a string containing the version of AK4953 driver.

Description

The DRV_AK4953_VersionStrGet function returns a string in the format: ".[.][]" Where: is the AK4953 driver's version number. is the AK4953 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces.

Remarks

None.

Preconditions

None.

Example 1

"0.03a" "1.00"

Example 2

```
int8_t *AK4953string;
AK4953string = DRV_AK4953_VersionStrGet();
```

Function

int8_t* DRV_AK4953_VersionStrGet(void)

DRV_AK4953_VolumeGet Function

This function gets the volume for AK4953 CODEC.

Implementation: Dynamic

File

drv_ak4953.h

C

```
uint8_t DRV_AK4953_VolumeGet(DRV_HANDLE handle, DRV_AK4953_CHANNEL chan);
```

Returns

None.

Description

This functions gets the current volume programmed to the CODEC AK4953.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

  volume = DRV_AK4953_VolumeGet(myAK4953Handle,DRV_AK4953_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set

Function

uint8_t DRV_AK4953_VolumeGet(DRV_HANDLE handle, DRV_AK4953_CHANNEL chan)

c) Other Functions

DRV_AK4953_BufferAddWrite Function

Schedule a non-blocking driver write operation.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_BufferAddWrite(const DRV_HANDLE handle, DRV_AK4953_BUFFER_HANDLE *
bufferHandle, void * buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4953_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4953_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4953_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4953_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4953 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4953 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 device instance and the DRV_AK4953_Status must have returned SYS_STATUS_READY.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK4953_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4953_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_BufferEventHandlerSet(myAK4953Handle,
                APP_AK4953BufferEventHandler, (uintptr_t)&myAppObj);
```

```
DRV_AK4953_BufferAddWrite(myAK4953handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4953_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4953BufferEventHandler(DRV_AK4953_BUFFER_EVENT event,
        DRV_AK4953_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4953 instance as return by the DRV_AK4953_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4953_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_AK4953_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

DRV_AK4953_BufferAddWriteRead Function

Schedule a non-blocking driver write-read operation.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_BufferAddWriteRead(const DRV_HANDLE handle, DRV_AK4953_BUFFER_HANDLE *
bufferHandle, void * transmitBuffer, void * receiveBuffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4953_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4953_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4953_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4953_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4953 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4953 driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every AK4953 write. The transmit and receive size must be same.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 device instance and the DRV_AK4953_Status must have returned SYS_STATUS_READY.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_AK4953_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_AK4953_BUFFER_HANDLE bufferHandle;

// myak4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_BufferEventHandlerSet(myak4953Handle,
                APP_AK4953BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4953_BufferAddWriteRead(myak4953handle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_AK4953_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
```

```
// the buffer is processed.

void APP_AK4953BufferEventHandler(DRV_AK4953_BUFFER_EVENT event,
        DRV_AK4953_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4953 instance as returned by the DRV_AK4953_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	The buffer where the transmit data will be stored
receiveBuffer	The buffer where the received data will be stored
size	Buffer size in bytes

Function

void DRV_AK4953_BufferAddWriteRead

(

const DRV_HANDLE handle,

DRV_AK4953_BUFFER_HANDLE *bufferHandle,

void *transmitBuffer,

void *receiveBuffer,

size_t size

)

DRV_AK4953_MuteOff Function

This function disables AK4953 output for soft mute.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_MuteOff(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables AK4953 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

    DRV_AK4953_MuteOff(myAK4953Handle); //AK4953 output soft mute disabled
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4953_MuteOff(DRV_HANDLE handle)

DRV_AK4953_MuteOn Function

This function allows AK4953 output for soft mute on.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_MuteOn(DRV_HANDLE handle);
```

Returns

None.

Description

This function Enables AK4953 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
```

```
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

DRV_AK4953_MuteOn(myAK4953Handle);  //AK4953 output soft muted
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4953_MuteOn(DRV_HANDLE handle);

DRV_AK4953_VolumeSet Function

This function sets the volume for AK4953 CODEC.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_VolumeSet(DRV_HANDLE handle, DRV_AK4953_CHANNEL channel, uint8_t volume);
```

Returns

None.

Description

This functions sets the volume value from 0-255. The codec has DAC value to volume range mapping as :- 00 H : +12dB FF H : -115dB In order to make the volume value to dB mapping monotonically increasing from 00 to FF, re-mapping is introduced which reverses the volume value to dB mapping as well as normalizes the volume range to a more audible dB range. The current driver implementation assumes that all dB values under -60 dB are inaudible to the human ear. Re-Mapped values 00 H : -60 dB FF H : +12 dB

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

    DRV_AK4953_VolumeSet(myAK4953Handle, DRV_AK4953_CHANNEL_LEFT, 120);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set
volume	volume value specified in the range 0-255 (0x00 to 0xFF)

Function

void DRV_AK4953_VolumeSet(DRV_HANDLE handle, DRV_AK4953_CHANNEL channel, uint8_t volume);

DRV_AK4953_BufferAddRead Function

Schedule a non-blocking driver read operation.

File

drv_ak4953.h

C

```
void DRV_AK4953_BufferAddRead(const DRV_HANDLE handle, DRV_AK4953_BUFFER_HANDLE * bufferHandle,
void * buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4953_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4953_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4953_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4953_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4953 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4953 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 device instance and the DRV_AK4953_Status must have returned SYS_STATUS_READY.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_AK4953_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4953 instance as return by the DRV_AK4953_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4953_BufferAddRead

(

const DRV_HANDLE handle,

DRV_AK4953_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

DRV_AK4953_IntExtMicSet Function

This function sets up the codec for the internal or the external microphone use.

File

drv_ak4953.h

C

```
void DRV_AK4953_IntExtMicSet(DRV_HANDLE handle, DRV_AK4953_INT_EXT_MIC micInput);
```

Returns

None

Description

This function sets up the codec for the internal or the external microphone use.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	Internal vs External mic input

Function

void DRV_AK4953_IntExtMicSet(DRV_HANDLE handle);

DRV_AK4953_MonoStereoMicSet Function

This function sets up the codec for the Mono or Stereo microphone mode.

File

drv_ak4953.h

C

```
void DRV_AK4953_MonoStereoMicSet(DRV_HANDLE handle, DRV_AK4953_MONO_STEREO_MIC mono_stereo_mic);
```

Returns

None

Description

This function sets up the codec for the Mono or Stereo microphone mode.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4953_MonoStereoMicSet(DRV_HANDLE handle);

d) Data Types and Constants

DRV_AK4953_AUDIO_DATA_FORMAT Enumeration

Identifies the Serial Audio data interface format.

File

drv_ak4953.h

C

```
typedef enum {
  DRV_AK4953_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_24BIT_LSB_SDTI = 0,
  DRV_AK4953_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_16BIT_LSB_SDTI,
  DRV_AK4953_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_24BIT_MSB_SDTI,
  DRV_AK4953_AUDIO_DATA_FORMAT_I2S
} DRV_AK4953_AUDIO_DATA_FORMAT;
```

Description

AK4953 Audio data format

This enumeration identifies Serial Audio data interface format.

DRV_AK4953_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_ak4953.h

C

```
typedef enum {
  DRV_AK4953_BUFFER_EVENT_COMPLETE,
  DRV_AK4953_BUFFER_EVENT_ERROR,
  DRV_AK4953_BUFFER_EVENT_ABORT
} DRV_AK4953_BUFFER_EVENT;
```

Members

Members	Description
DRV_AK4953_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_AK4953_BUFFER_EVENT_ERROR	Error while processing the request
DRV_AK4953_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

AK4953 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_AK4953_BufferAddWrite() function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_AK4953_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_AK4953_BUFFER_EVENT_HANDLER Type

Pointer to a AK4953 Driver Buffer Event handler function

File

drv_ak4953.h

C

```
typedef void (* DRV_AK4953_BUFFER_EVENT_HANDLER)(DRV_AK4953_BUFFER_EVENT event,
DRV_AK4953_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

AK4953 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK4953 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_AK4953_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK4953_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_AK4953_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4953_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver (I2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_AK4953_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_AK4953_BUFFER_EVENT event,
                               DRV_AK4953_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
```

```
            break;
        }
    }
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4953_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

File

drv_ak4953.h

C

```
typedef uintptr_t DRV_AK4953_BUFFER_HANDLE;
```

Description

AK4953 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK4953_BufferAddWrite() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_AK4953_COMMAND_EVENT_HANDLER Type

Pointer to a AK4953 Driver Command Event Handler Function

File

drv_ak4953.h

C

```
typedef void (* DRV_AK4953_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

Returns

None.

Description

AK4953 Driver Command Event Handler Function

This data type defines the required function signature for the AK4953 driver command event handling callback function.

A command is a control instruction to the AK4953 CODEC. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4953_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_AK4953CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4953_DIGITAL_BLOCK_CONTROL Enumeration

Identifies Bass-Boost Control function

File

drv_ak4953.h

C

```
typedef enum {
  DRV_AK4953_RECORDING_MODE,
  DRV_AK4953_PLAYBACK_MODE,
  DRV_AK4953_RECORDING_PLAYBACK_2_MODE,
  DRV_AK4953_LOOPBACK_MODE
} DRV_AK4953_DIGITAL_BLOCK_CONTROL;
```

Members

Members	Description
DRV_AK4953_RECORDING_MODE	This is the default setting
DRV_AK4953_PLAYBACK_MODE	Min control
DRV_AK4953_RECORDING_PLAYBACK_2_MODE	Medium control
DRV_AK4953_LOOPBACK_MODE	Maximum control

Description

AK4953 Bass-Boost Control

This enumeration identifies the settings for Bass-Boost Control function.

Remarks

None.

DRV_AK4953_INIT Structure

Defines the data required to initialize or reinitialize the AK4953 driver

File

drv_ak4953.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX i2sDriverModuleIndex;
  SYS_MODULE_INDEX i2cDriverModuleIndex;
  uint32_t samplingRate;
  uint8_t volume;
  DRV_AK4953_AUDIO_DATA_FORMAT audioDataFormat;
} DRV_AK4953_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX i2sDriverModuleIndex;	Identifies data module(I2S) driver ID for data interface of CODEC
SYS_MODULE_INDEX i2cDriverModuleIndex;	Identifies data module(I2C) driver ID for control interface of CODEC
uint32_t samplingRate;	Sampling rate
uint8_t volume;	Volume
DRV_AK4953_AUDIO_DATA_FORMAT audioDataFormat;	Identifies the Audio data format

Description

AK4953 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK4953 CODEC driver.

Remarks

None.

_DRV_AK4953_H Macro

File

drv_ak4953.h

C

```
#define _DRV_AK4953_H
```

Description

Include files.

DRV_AK4953_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_ak4953.h

C

```
#define DRV_AK4953_BUFFER_HANDLE_INVALID ((DRV_AK4953_BUFFER_HANDLE)(-1))
```

Description

AK4953 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_AK4953_BufferAddWrite() function if the buffer add request was not successful.

Remarks

None

DRV_AK4953_COUNT Macro

Number of valid AK4953 driver indices

File

drv_ak4953.h

C

```
#define DRV_AK4953_COUNT
```

Description

AK4953 Driver Module Count

This constant identifies the maximum number of AK4953 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of AK4953 instances on this microcontroller.

Remarks

This value is part-specific.

DRV_AK4953_INDEX_0 Macro

AK4953 driver index definitions

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_0 0
```

Description

Driver AK4953 Module Index

These constants provide AK4953 driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_AK4953_Initialize and DRV_AK4953_Open routines to identify the driver instance in use.

DRV_AK4953_INDEX_1 Macro

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_1 1
```

Description

This is macro DRV_AK4953_INDEX_1.

DRV_AK4953_INDEX_2 Macro

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_2 2
```

Description

This is macro DRV_AK4953_INDEX_2.

DRV_AK4953_INDEX_3 Macro

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_3 3
```

Description

This is macro DRV_AK4953_INDEX_3.

DRV_AK4953_INDEX_4 Macro

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_4 4
```

Description

This is macro DRV_AK4953_INDEX_4.

DRV_AK4953_INDEX_5 Macro

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_5 5
```

Description

This is macro DRV_AK4953_INDEX_5.

DRV_I2C_INDEX Macro

File

drv_ak4953.h

C

```
#define DRV_I2C_INDEX DRV_AK4953_I2C_INSTANCES_NUMBER
```

Description

This is macro DRV_I2C_INDEX.

DRV_AK4953_CHANNEL Enumeration

Identifies Left/Right Audio channel

File

drv_ak4953.h

C

```
typedef enum {
  DRV_AK4953_CHANNEL_LEFT,
  DRV_AK4953_CHANNEL_RIGHT,
  DRV_AK4953_CHANNEL_LEFT_RIGHT,
  DRV_AK4953_NUMBER_OF_CHANNELS
} DRV_AK4953_CHANNEL;
```

Description

AK4953 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

DRV_AK4953_INT_EXT_MIC Enumeration

Identifies the Mic input source.

File

drv_ak4953.h

C

```
typedef enum {
  INT_MIC,
  EXT_MIC
} DRV_AK4953_INT_EXT_MIC;
```

Description

AK4953 Mic Internal / External Input

This enumeration identifies the Mic input source.

DRV_AK4953_MONO_STEREO_MIC Enumeration

Identifies the Mic input as Mono / Stereo.

File

drv_ak4953.h

C

```
typedef enum {
  ALL_ZEROS,
  MONO_RIGHT_CHANNEL,
  MONO_LEFT_CHANNEL,
  STEREO
} DRV_AK4953_MONO_STEREO_MIC;
```

Description

AK4953 Mic Mono / Stereo Input

This enumeration identifies the Mic input as Mono / Stereo.

Files

Files

Name	Description
drv_ak4953.h	AK4953 CODEC Driver Interface header file
drv_ak4953_config_template.h	AK4953 Codec Driver Configuration Template.

Description

This section lists the source and header files used by the AK4953Codec Driver Library.

drv_ak4953.h

AK4953 CODEC Driver Interface header file

Enumerations

	Name	Description
	DRV_AK4953_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4953_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4953_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4953_DIGITAL_BLOCK_CONTROL	Identifies Bass-Boost Control function
	DRV_AK4953_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4953_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.

Functions

	Name	Description
	DRV_AK4953_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4953_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_AK4953_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4953_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_AK4953_Close	Closes an opened-instance of the AK4953 driver. **Implementation:** Dynamic
	DRV_AK4953_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. **Implementation:** Dynamic
	DRV_AK4953_Deinitialize	Deinitializes the specified instance of the AK4953 driver module. **Implementation:** Dynamic
	DRV_AK4953_Initialize	Initializes hardware and data for the instance of the AK4953 DAC module. **Implementation:** Dynamic
	DRV_AK4953_IntExtMicSet	This function sets up the codec for the internal or the external microphone use.
	DRV_AK4953_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4953_MuteOff	This function disables AK4953 output for soft mute. **Implementation:** Dynamic
	DRV_AK4953_MuteOn	This function allows AK4953 output for soft mute on. **Implementation:** Dynamic
	DRV_AK4953_Open	Opens the specified AK4953 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_AK4953_SamplingRateGet	This function gets the sampling rate set on the DAC AK4953. **Implementation:** Dynamic

	DRV_AK4953_SamplingRateSet	This function sets the sampling rate of the media stream. **Implementation:** Dynamic
	DRV_AK4953_Status	Gets the current status of the AK4953 driver module. **Implementation:** Dynamic
	DRV_AK4953_Tasks	Maintains the driver's control and data interface state machine. **Implementation:** Dynamic
	DRV_AK4953_VersionGet	This function returns the version of AK4953 driver. **Implementation:** Dynamic
	DRV_AK4953_VersionStrGet	This function returns the version of AK4953 driver in string format. **Implementation:** Dynamic
	DRV_AK4953_VolumeGet	This function gets the volume for AK4953 CODEC. **Implementation:** Dynamic
	DRV_AK4953_VolumeSet	This function sets the volume for AK4953 CODEC. **Implementation:** Dynamic

Macros

	Name	Description
	_DRV_AK4953_H	Include files.
	DRV_AK4953_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4953_COUNT	Number of valid AK4953 driver indices
	DRV_AK4953_INDEX_0	AK4953 driver index definitions
	DRV_AK4953_INDEX_1	This is macro DRV_AK4953_INDEX_1.
	DRV_AK4953_INDEX_2	This is macro DRV_AK4953_INDEX_2.
	DRV_AK4953_INDEX_3	This is macro DRV_AK4953_INDEX_3.
	DRV_AK4953_INDEX_4	This is macro DRV_AK4953_INDEX_4.
	DRV_AK4953_INDEX_5	This is macro DRV_AK4953_INDEX_5.
	DRV_I2C_INDEX	This is macro DRV_I2C_INDEX.

Structures

	Name	Description
	DRV_AK4953_INIT	Defines the data required to initialize or reinitialize the AK4953 driver

Types

	Name	Description
	DRV_AK4953_BUFFER_EVENT_HANDLER	Pointer to a AK4953 Driver Buffer Event handler function
	DRV_AK4953_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4953_COMMAND_EVENT_HANDLER	Pointer to a AK4953 Driver Command Event Handler Function

Description

AK4953 CODEC Driver Interface

The AK4953 CODEC device driver interface provides a simple interface to manage the AK4953 106dB 192kHz 24-Bit DAC that can be interfaced Microchip Microcontroller. This file provides the interface definition for the AK4953 CODEC device driver.

File Name

drv_AK4953.h

Company

Microchip Technology Inc.

drv_ak4953_config_template.h

AK4953 Codec Driver Configuration Template.

Macros

	Name	Description
	DRV_AK4953_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4953_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4953_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4953_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4953_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.
	DRV_AK4953_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.

Description

AK4953 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_ak4953_config_template.h

Company

Microchip Technology Inc.

Comparator Driver Library

This topic describes the Comparator Driver Library.

Introduction

The Comparator Static Driver provides a high-level interface to manage the Comparator module on the Microchip family of microcontrollers.

Description

Through MHC, this driver provides an API to initialize the Comparator module, as well as reference channels, CVREF, inputs, and interrupts.

Library Interface

Function(s)

	Name	Description
	DRV_CMP_Initialize	Initializes the Comparator instance for the specified driver index. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the Comparator Driver Library.

Function(s)

DRV_CMP_Initialize Function

Initializes the Comparator instance for the specified driver index.

Implementation: Static

File

help_drv_cmp.h

C

```
void DRV_CMP_Initialize();
```

Returns

None.

Description

This routine initializes the Comparator driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters. The driver instance index is independent of the Comparator module ID. For example, driver instance 0 can be assigned to Comparator 2.

Remarks

This routine must be called before any other Comparator routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_CMP_Initialize(void)

CPLD XC2C64A Driver Library

This topic describes the CPLD XC2C64A Driver Library.

Introduction

This library provides an interface to manage the CPLD XC2C64A devices on Microchip starter kits.

Description

A CPLD is provided on the Multimedia Expansion Board (MEB), which can be used to configure the graphics controller bus interface, SPI channel and Chip Selects used for SPI Flash, the MRF24WBOMA, and the expansion slot. The general I/O inputs are used to change the configuration, which can be done at run-time.

Specific CPLD configuration information is available in the *"Multimedia Expansion Board (MEB) User's Guide"* (DS60001160), which is available from the MEB product page:
http://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=DM320005

Using the Library

This topic describes the basic architecture of the CPLD XC2C64A Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_xc2c64a.h`

The interface to the CPLD XC2C64A Driver Library is defined in the `drv_xc2c64a.h` header file. Any C language source (`.c`) file that uses the CPLD XC2C64A Driver library should include this header.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the CPLD XC2C64A Driver.

Library Interface Section	Description
Functions	Provides CPLD XC2C64A initialization and configuration functions.

Configuring the Library

The configuration of the CPLD XC2C64A Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the CPLD XC2C64A Driver. Based on the selections made, the CPLD XC2C64A may support the selected features. These configuration settings will apply to all instances of the CPLD XC2C64A Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

Building the Library

This section lists the files that are available in the CPLD XC2C64A Driver Library.

Description

This section list the files that are available in the /src folder of the CPLD XC2C64A Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/driver/cpld/xc2c64a.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
/drv_xc2c64a.h	Header file that exports the CPLD XC2C64A Driver API.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_xc2c64a.c	Basic CPLD XC2C64A Driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The CPLD XC2C64A Driver Library is not dependent on other modules.

Library Interface

a) Functions

	Name	Description
	CPLDGetDeviceConfiguration	Returns the selected device. **Implementation:** Static
	CPLDGetGraphicsConfiguration	Returns the selected PMP bus, 8 or 16-bit, interface to the graphics controller. **Implementation:** Static
	CPLDGetSPIConfiguration	Returns the selected SPI Channel. **Implementation:** Static
	CPLDInitialize	Initializes the control I/O to the CPLD and places the CPLD in a known state. **Implementation:** Static
	CPLDSetGraphicsConfiguration	Selects the PMP bus, 8 or 16-bit, interface to the graphic controller. **Implementation:** Static
	CPLDSetSPIFlashConfiguration	Selects the SPI Flash device. **Implementation:** Static
	CPLDSetWiFiConfiguration	Selects the Wi-Fi device. **Implementation:** Static
	CPLDSetZigBeeConfiguration	Selects the ZigBee/MiWi device. **Implementation:** Static

b) Data Types and Constants

	Name	Description
	CPLD_DEVICE_CONFIGURATION	CPLD device configuration.
	CPLD_GFX_CONFIGURATION	CPLD graphics controller PMP bus configuration.
	CPLD_SPI_CONFIGURATION	CPLD SPI channel selection.

Description

This section describes the API functions of the CPLD XC2C64A Driver Library.

Refer to each section for a detailed description.

a) Functions

CPLDGetDeviceConfiguration Function

Returns the selected device.

Implementation: Static

File

drv_xc2c64a.h

C

```
CPLD_DEVICE_CONFIGURATION CPLDGetDeviceConfiguration();
```

Returns

- CPLD_DEVICE_SPI_FLASH - SPI Flash.
- CPLD_DEVICE_WiFi - Zero G 802.11 Wi-Fi.
- CPLD_DEVICE_ZIGBEE - ZigBee/MiWi.

Description

This routine returns the selected CPLD device.

Remarks

None.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

```
// Initialize the CPLD
CPLDInitialize();

if(CPLDGetDeviceConfiguration() != CPLD_DEVICE_SPI_FLASH)
{
    // error - not setup as default

}
```

Function

CPLD_DEVICE_CONFIGURATION CPLDGetDeviceConfiguration(void)

CPLDGetGraphicsConfiguration Function

Returns the selected PMP bus, 8 or 16-bit, interface to the graphics controller.

Implementation: Static

File

drv_xc2c64a.h

C

```
CPLD_GFX_CONFIGURATION CPLDGetGraphicsConfiguration();
```

Returns

- CPLD_GFX_CONFIG_8BIT - Graphics controller is configured for 8-bit PMP data bus interface.
- CPLD_GFX_CONFIG_16BIT - Graphics controller is configured for 16-bit PMP data bus interface.

Description

This routine gets the configuration of the PMP, 8 or 16-bit, data bus interface.

Remarks

None.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

```
// Initialize the CPLD
CPLDInitialize();

if(CPLDGetGraphicsConfiguration() != CPLD_GFX_CONFIG_8BIT)
{
    // error - not setup as default
}
```

Function

CPLD_GFX_CONFIGURATION CPLDGetGraphicsConfiguration(void)

CPLDGetSPIConfiguration Function

Returns the selected SPI Channel.

Implementation: Static

File

drv_xc2c64a.h

C

```
CPLD_SPI_CONFIGURATION CPLDGetSPIConfiguration();
```

Returns

- CPLD_SPI2A - SPI Channel 2A with chip select PORT G bit 9 and external interrupt 1 or 3
- CPLD_SPI3A - SPI Channel 3A with chip select PORT F bit 12 and change notice 9
- CPLD_SPI2 - SPI Channel 2 with chip select PORT G bit 9 and external interrupt 1 or 3

Description

This routine returns the selected SPI channel.

Remarks

SPI channels 2 and 2A are located on the same pins. SPI channels 2A and 3A are only available on PIC32MX5xx/6xx/7xx series parts.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

```
// Initialize the CPLD
CPLDInitialize();

if(CPLDGetSPIConfiguration() != CPLD_SPI2A)
{
    // error - not setup as default

}
```

Function

CPLD_SPI_CONFIGURATION CPLDGetSPIConfiguration(void)

CPLDInitialize Function

Initializes the control I/O to the CPLD and places the CPLD in a known state.

Implementation: Static

File

drv_xc2c64a.h

C

```
void CPLDInitialize();
```

Returns

None.

Description

This routine configures the control I/O and places the CPLD in a known state.

- Graphics Controller Bus - 8-bit PMP data interface.
- SPI Channel - SPI2/SPI2A.
- Chip Select - PORT G bit 9.
- External Interrupt 1 or 3
- Device - SPI Flash.

Remarks

None.

Preconditions

None.

Example

```
// Initialize the CPLD
CPLDInitialize();

// CPLD is configured in the default state
```

Function

void CPLDInitialize(void)

CPLDSetGraphicsConfiguration Function

Selects the PMP bus, 8 or 16-bit, interface to the graphic controller.

Implementation: Static

File

drv_xc2c64a.h

C

```
void CPLDSetGraphicsConfiguration(CPLD_GFX_CONFIGURATION configuration);
```

Returns

None.

Description

This routine sets the configuration pins on the graphics controller to select between an 8 or 16-bit data bus interface.

Remarks

The graphics controller interface configuration must be done before initializing the graphics controller.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

Setting the graphics controller to a 16-bit interface

```
// Initialize the CPLD
CPLDInitialize();

// configure the graphics controller for a 16-bit PMP interface.
CPLDSetGraphicsConfiguration(CPLD_GFX_CONFIG_16BIT);
```

Setting the graphics controller to a 8-bit interface

```
// Initialize the CPLD
CPLDInitialize();

// configure the graphics controller for a 8-bit PMP interface.
CPLDSetGraphicsConfiguration(CPLD_GFX_CONFIG_8BIT);
```

Parameters

Parameters	Description
configuration	the type of interface configuration.

Function

void CPLDSetGraphicsConfiguration(CPLD_GFX_CONFIGURATION configuration)

CPLDSetSPIFlashConfiguration Function

Selects the SPI Flash device.

Implementation: Static

File

drv_xc2c64a.h

C

```
void CPLDSetSPIFlashConfiguration(CPLD_SPI_CONFIGURATION configuration);
```

Returns

None.

Description

This routine configures the CPLD to communicate to the SPI Flash device with the selected SPI channel and Chip Select.

Remarks

SPI channels 2 and 2A are located on the same pins. SPI channels 2A and 3A are only available on PIC32MX5xx/6xx/7xx series parts.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

Setting CPLD to SPI Flash using SPI channel 2 and chip select PORT G bit 9

```
// Initialize the CPLD
CPLDInitialize();

// configure the SPI Flash to use SPI channel 2 and chip select PORT G bit 9
CPLDSetSPIFlashConfiguration(CPLD_SPI2);
```

Setting CPLD to SPI Flash using SPI channel 2A and chip select PORT G bit 9

```
// Initialize the CPLD
CPLDInitialize();

// configure the SPI Flash to use SPI channel 2A and chip select PORT G bit 9
CPLDSetSPIFlashConfiguration(CPLD_SPI2A);
```

Setting CPLD to SPI Flash using SPI channel 3A and chip select PORT F bit 12

```
// Initialize the CPLD
CPLDInitialize();

// configure the SPI Flash to use SPI channel 3A and chip select PORT F bit 12
CPLDSetSPIFlashConfiguration(CPLD_SPI3A);
```

Parameters

Parameters	Description
configuration	the type of SPI channel used by the SPI Flash device.

Function

void CPLDSetSPIFlashConfiguration(CPLD_SPI_CONFIGURATION configuration)

CPLDSetWiFiConfiguration Function

Selects the Wi-Fi device.

Implementation: Static

File

drv_xc2c64a.h

C

```
void CPLDSetWiFiConfiguration(CPLD_SPI_CONFIGURATION configuration);
```

Returns

None.

Description

This routine configures the CPLD to communicate to the Wi-Fi device with the selected SPI channel, chip select and external interrupt or change notice.

Remarks

SPI channels 2 and 2A are located on the same pins. SPI channels 2A and 3A are only available on PIC32MX5xx/6xx/7xx series parts.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

Setting CPLD to Wi-Fi using SPI channel 2, chip select PORT G bit 9 and external interrupt 3

```
// Initialize the CPLD
CPLDInitialize();

// configure the Wi-Fi to use SPI channel 2, chip select PORT G bit 9 and external interrupt 3
CPLDSetWiFiConfiguration(CPLD_SPI2);
```

Setting CPLD to Wi-Fi using SPI channel 2A, chip select PORT G bit 9 and external interrupt 3

```
// Initialize the CPLD
CPLDInitialize();

// configure the Wi-Fi to use SPI channel 2A, chip select PORT G bit 9 and external interrupt 3
CPLDSetWiFiConfiguration(CPLD_SPI2A);
```

Setting CPLD to Wi-Fi using SPI channel 3A, chip select PORT F bit 12 and change notice 9

```
// Initialize the CPLD
CPLDInitialize();

// configure the Wi-Fi to use SPI channel 3A, chip select PORT F bit 12 and change notice 9
CPLDSetWiFiConfiguration(CPLD_SPI3A);
```

Parameters

Parameters	Description
configuration	the type of SPI channel used by the Wi-Fi device.

Function

void CPLDSetWiFiConfiguration(CPLD_SPI_CONFIGURATION configuration)

CPLDSetZigBeeConfiguration Function

Selects the ZigBee/MiWi device.

Implementation: Static

File

drv_xc2c64a.h

C

```
void CPLDSetZigBeeConfiguration(CPLD_SPI_CONFIGURATION configuration);
```

Returns

None.

Description

This routine configures the CPLD to communicate to the ZigBee/MiWi device with the selected SPI channel, chip select and external interrupt or change notice.

Remarks

SPI channels 2 and 2A are located on the same pins. SPI channels 2A and 3A are only available on PIC32MX5xx/6xx/7xx series parts.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

Setting CPLD to ZigBee/MiWi using SPI channel 2, chip select PORT G bit 9 and external interrupt 3

```
// Initialize the CPLD
CPLDInitialize();

// configure the ZigBee/MiWi to use SPI channel 2, chip select PORT G bit 9 and external interrupt 3
CPLDSetZigBeeConfiguration(CPLD_SPI2);
```

Setting CPLD to ZigBee/MiWi using SPI channel 2A, chip select PORT G bit 9 and external interrupt 3

```
// Initialize the CPLD
CPLDInitialize();

// configure the ZigBee/MiWi to use SPI channel 2A, chip select PORT G bit 9 and external interrupt 3
CPLDSetZigBeeConfiguration(CPLD_SPI2A);
```

Setting CPLD to ZigBee/MiWi using SPI channel 3A, chip select PORT F bit 12 and change notice 9

```
// Initialize the CPLD
CPLDInitialize();

// configure the ZigBee/MiWi to use SPI channel 3A, chip select PORT F bit 12 and change notice 9
CPLDSetZigBeeConfiguration(CPLD_SPI3A);
```

Parameters

Parameters	Description
configuration	the type of SPI channel used by the ZigBee/MiWi device.

Function

void CPLDSetZigBeeConfiguration(CPLD_SPI_CONFIGURATION configuration)

b) Data Types and Constants

CPLD_DEVICE_CONFIGURATION Enumeration

CPLD device configuration.

File

drv_xc2c64a.h

C

```
typedef enum {
  CPLD_DEVICE_SPI_FLASH,
  CPLD_DEVICE_WiFi,
  CPLD_DEVICE_ZIGBEE
} CPLD_DEVICE_CONFIGURATION;
```

Members

Members	Description
CPLD_DEVICE_SPI_FLASH	SPI Flash
CPLD_DEVICE_WiFi	Zero G Wi-Fi
CPLD_DEVICE_ZIGBEE	ZigBee/MiWi

Description

The CPLD can be configured to communicate to three different devices. The application may call routine, CPLDGetDeviceConfiguration, to obtain what device the CPLD is configured to communicate with.

Remarks

None.

Example

```
// select 16-bit PMP data bus
if(CPLDGetDeviceConfiguration() != CPLD_DEVICE_SPI_FLASH)
{
    // error - not default configuration
}
```

CPLD_GFX_CONFIGURATION Enumeration

CPLD graphics controller PMP bus configuration.

File

drv_xc2c64a.h

C

```
typedef enum {
  CPLD_GFX_CONFIG_8BIT,
  CPLD_GFX_CONFIG_16BIT
} CPLD_GFX_CONFIGURATION;
```

Members

Members	Description
CPLD_GFX_CONFIG_8BIT	Configure the Graphics Controller to use 8-bit PMP data bus
CPLD_GFX_CONFIG_16BIT	Configure the Graphics Controller to use 16-bit PMP data bus

Description

The application can select what PMP bus configuration, 8 or 16-bit data bus, when interfacing with the graphics controller.

Remarks

None.

Example

```
// select 16-bit PMP data bus
CPLDSetGraphicsConfiguration(CPLD_GFX_CONFIG_16BIT);
```

CPLD_SPI_CONFIGURATION Enumeration

CPLD SPI channel selection.

File

drv_xc2c64a.h

C

```
typedef enum {
  CPLD_SPI2A,
  CPLD_SPI3A,
  CPLD_SPI2
} CPLD_SPI_CONFIGURATION;
```

Members

Members	Description
CPLD_SPI2A	PIC32 SPI Channel 2A and chip select PORT G bit 9
CPLD_SPI3A	PIC32 SPI Channel 3A and chip select PORT F bit 12
CPLD_SPI2	PIC32 SPI Channel 2 and chip select PORT G bit 9

Description

The application can select what SPI channel will be used as the communication interface. It will also select the Chip Select use for the device.

Remarks

Only one SPI channel can be select for a device. SPI channels 2 and 2A are located on the same pins. SPI channels 2A and 3A are only available on PIC32MX5xx/6xx/7xx series devices.

Example

```
// select SPI channel two for SPI Flash
CPLDSetSPIFlashConfiguration(CPLD_SPI2);
```

Files

Files

Name	Description
drv_xc2c64a.h	This file contains the interface definition for the CUPLD controller.

Description

This section lists the source and header files used by the SPI Flash Driver Library.

drv_xc2c64a.h

This file contains the interface definition for the CUPLD controller.

Enumerations

	Name	Description
	CPLD_DEVICE_CONFIGURATION	CPLD device configuration.
	CPLD_GFX_CONFIGURATION	CPLD graphics controller PMP bus configuration.
	CPLD_SPI_CONFIGURATION	CPLD SPI channel selection.

Functions

	Name	Description
	CPLDGetDeviceConfiguration	Returns the selected device. **Implementation:** Static
	CPLDGetGraphicsConfiguration	Returns the selected PMP bus, 8 or 16-bit, interface to the graphics controller. **Implementation:** Static
	CPLDGetSPIConfiguration	Returns the selected SPI Channel. **Implementation:** Static
	CPLDInitialize	Initializes the control I/O to the CPLD and places the CPLD in a known state. **Implementation:** Static
	CPLDSetGraphicsConfiguration	Selects the PMP bus, 8 or 16-bit, interface to the graphic controller. **Implementation:** Static
	CPLDSetSPIFlashConfiguration	Selects the SPI Flash device. **Implementation:** Static
	CPLDSetWiFiConfiguration	Selects the Wi-Fi device. **Implementation:** Static
	CPLDSetZigBeeConfiguration	Selects the ZigBee/MiWi device. **Implementation:** Static

Description

CUPLD Controller Interface File.

This library provides a low-level abstraction of the CUPLD device. It can be used to simplify low-level access to the device without the necessity of interacting directly with the communication module's registers, thus hiding differences from one serial device variant to another.

File Name

drv_xc2c64a.h

Company

Microchip Technology Inc.

EBI Driver Library

This topic describes the EBI Driver Library.

Introduction

The External Bus Interface Static Driver provides a high-level interface to manage the External Bus Interface module on the Microchip family of microcontrollers.

Description

Through the MHC, this driver provides an API to initialize the EBI module, as well as Chip Selects, timing parameters, output signals, and memory characteristics.

Library Interface

Function(s)

	Name	Description
	DRV_EBI_Initialize	Initializes the External Bus Interface instance for the specified driver index. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the EBI Driver Library.

Function(s)

DRV_EBI_Initialize Function

Initializes the External Bus Interface instance for the specified driver index.

Implementation: Static

File

help_drv_ebi.h

C

```
void DRV_EBI_Initialize();
```

Returns

None.

Description

This routine initializes the External Bus Interface driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

This routine must be called before any other External Bus Interface routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_EBI_Initialize(void)

ENCx24J600 Driver Library Help

This section provides information on the ENCx24J600 Driver Library.

Introduction

This library provides a driver-level abstraction of the ENCx24J600 Ethernet MAC that can be connected to the PIC32. The driver implements the virtual MAC driver model that the MPLAB Harmony TCP/IP Stack requires. Please see the TCP/IP Stack Library MAC Driver Module for details.

The "Host-To-Network"_layer of a TCP/IP stack organization covers the Data Link and Physical Layers of the standard OSI stack. The Ethernet Controller provides the Data Link or Media Access Control Layer, in addition to other functions discussed in this section.

Description

The ENCx24J600 External MAC is an external module to the PIC32 that is connected through a SPI or PSP interface. This driver interfaces with the SPI driver to communicate with the external device to implement a complete Ethernet node in a system.

The following are some of the key features of this module:

- Supports 10/100 Ethernet
  - Full-Duplex and Half-Duplex operation
  - Broadcast, Multicast and Unicast packets
  - Manual and automatic flow control
  - Supports Auto-MDIX
- Fully configurable interrupts
- Configurable receive packet filtering using:
  - 64-bit Hash Table
  - 64-byte Pattern Match
  - Magic Packet™ Filtering
  - Runt Packet Detection and Filtering
- Supports Packet Payload Checksum calculation
- CRC Check
- Supports SPI interface

© 2013-2015 Microchip Technology Inc.

Using the Library

This topic describes the basic architecture and functionality of the Ethernet MAC driver and is meant for advanced users or TCP/IP stack driver developers.

Description

The user of this driver is the MPLAB Harmony TCP/IP stack. This Ethernet driver is not intended as a system-wide driver that the application or other system modules may use. It is intended for the sole use of the MPLAB Harmony TCP/IP stack and implements the virtual MAC model required by the stack.

Interface Header File: `drv_encx24j600.h`

The interface to the ENCx24J600 Driver Library is defined in the `drv_encx24j600.h` header file. Any C language source (`.c`) file that uses the ENCx24J600 Driver Library should include `drv_encx24j600.h`.

Library File: The ENCx24J600 Driver Library archive (`.a`) file is installed with MPLAB Harmony.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

The ENCx24J600 Driver Library provides the low-level abstraction of the communications protocol to communicate to the ENCx24J600 external MAC though the SPI peripheral on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the ENCx24J600 Driver Library interface.

Description

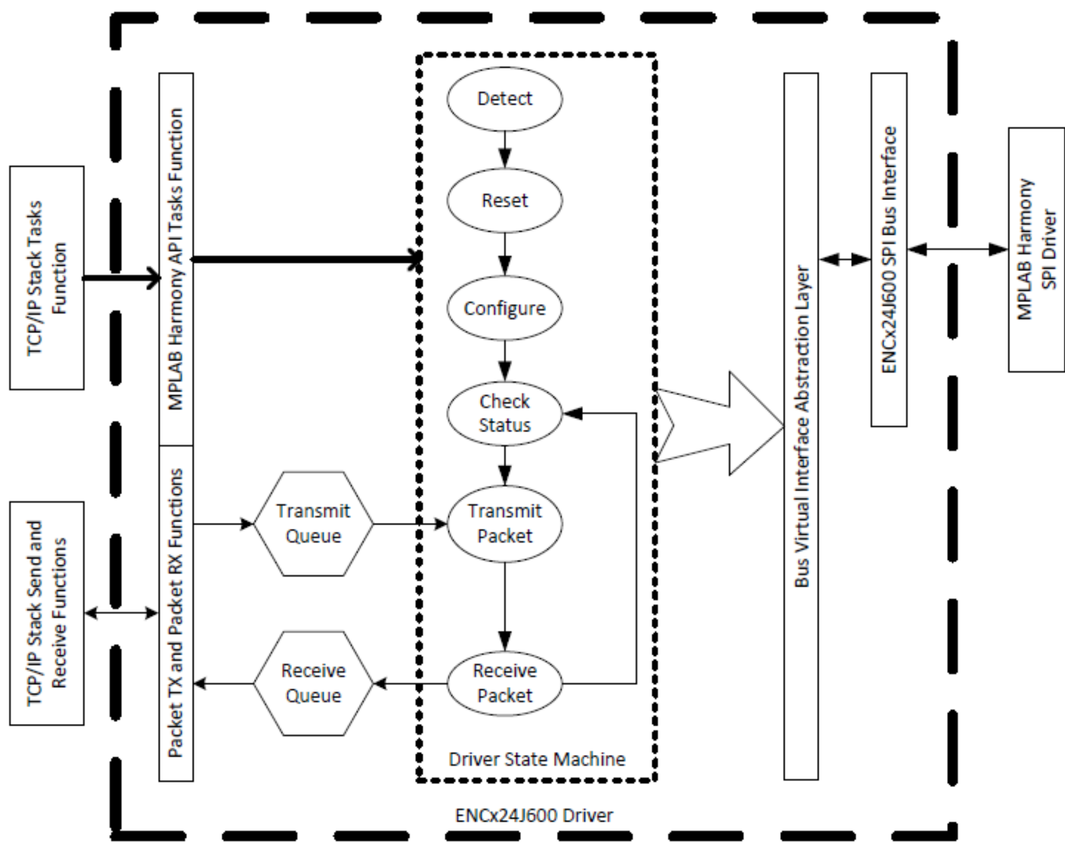
The ENCx24J600 Driver library has several different layers to it, as illustrated in the following figure. The interface layer has two main sections that are used the most often: The Tasks function, and the TCP/IP Send and Receive functions.

The Tasks function manages the internal state machine which detects, resets, and then configures the ENCx24J600 External MAC. It also handles the monitoring of the hardware status, sending and receiving packets.

The TCP/IP Send and Receive functions interact with the RAM-based queue of packets that are queued to send and packets that have been queued waiting for pick-up by the stack.

The main state machine does not interface directly to the SPI bus, but instead, interfaces to a virtual bus abstraction layer that allows for the replacement of the specific underlying bus implementation.

Abstraction Model



Library Overview

Refer to the section Driver Overview for how the driver operates in a system.

The library interface routines are divided into various sub-sections, each sub-section addresses one of the blocks or the overall operation of the ENCx24J600 Driver Library.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Data Transfer Functions	Provides data transfer functions available in the configuration.
Status Functions	Provides status functions.
Miscellaneous Functions	Provides miscellaneous driver functions.

How the Library Works

The library provides interfaces to support the TCP/IP virtual MAC interface.

Configuring the SPI Driver

This section describes the configuration settings for the ENCx24J600 Driver Library.

Description

Configuration

The ENC hardware requires a specific configuration of the SPI driver to work correctly. Inside the MHC SPI Driver configuration be sure to select:

- SPI clock rate of 14000000 or less. With a PB clock of 80 MHz, 13333333 is the clock rate.
- Clock mode of DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL
- Input phase of SPI_INPUT_SAMPLING_PHASE_AT_END

Recommended Settings

- Interrupt Driver mode
- Enhanced Buffer mode
- DMA mode enabled:
  - DMA block transfer size of at least 1600 bytes
  - Size of DMA buffer for dummy data of at least 1600 bytes
  - Ensure when setting up DMA in interrupt mode that the DMA interrupts are a higher priority than the SPI Driver interrupt

Example:

```
/*** SPI Driver Static Allocation Options ***/
#define DRV_SPI_INSTANCES_NUMBER            1
#define DRV_SPI_CLIENTS_NUMBER              1
#define DRV_SPI_ELEMENTS_PER_QUEUE          30

/*** SPI Driver DMA Options ***/
#define DRV_SPI_DMA_TXFER_SIZE              2048
#define DRV_SPI_DMA_DUMMY_BUFFER_SIZE       2048

/* SPI Driver Instance 0 Configuration */
#define DRV_SPI_SPI_ID_IDX0              SPI_ID_1
#define DRV_SPI_TASK_MODE_IDX0           DRV_SPI_TASK_MODE_ISR
#define DRV_SPI_SPI_MODE_IDX0            DRV_SPI_MODE_MASTER
#define DRV_SPI_ALLOW_IDLE_RUN_IDX0      false
#define DRV_SPI_SPI_PROTOCOL_TYPE_IDX0   DRV_SPI_PROTOCOL_TYPE_STANDARD
#define DRV_SPI_SPI_PROTOCOL_TYPE_IDX0   DRV_SPI_PROTOCOL_TYPE_STANDARD
#define DRV_SPI_COMM_WIDTH_IDX0          SPI_COMMUNICATION_WIDTH_8BITS
#define DRV_SPI_SPI_CLOCK_IDX0           CLK_BUS_PERIPHERAL_2
#define DRV_SPI_BAUD_RATE_IDX0           13333333
#define DRV_SPI_BUFFER_TYPE_IDX0         DRV_SPI_BUFFER_TYPE_ENHANCED
#define DRV_SPI_CLOCK_MODE_IDX0          DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL
#define DRV_SPI_INPUT_PHASE_IDX0         SPI_INPUT_SAMPLING_PHASE_AT_END
#define DRV_SPI_TX_INT_SOURCE_IDX0       INT_SOURCE_SPI_1_TRANSMIT
#define DRV_SPI_RX_INT_SOURCE_IDX0       INT_SOURCE_SPI_1_RECEIVE
#define DRV_SPI_ERROR_INT_SOURCE_IDX0    INT_SOURCE_SPI_1_ERROR
#define DRV_SPI_INT_VECTOR_IDX0          INT_VECTOR_SPI1
#define DRV_SPI_INT_PRIORITY_IDX0        INT_PRIORITY_LEVEL1
#define DRV_SPI_INT_SUB_PRIORITY_IDX0    INT_SUBPRIORITY_LEVEL0
#define DRV_SPI_QUEUE_SIZE_IDX0          30
#define DRV_SPI_RESERVED_JOB_IDX0        1
#define DRV_SPI_TX_DMA_CHANNEL_IDX0      DMA_CHANNEL_1
#define DRV_SPI_TX_DMA_THRESHOLD_IDX0    16
#define DRV_SPI_RX_DMA_CHANNEL_IDX0      DMA_CHANNEL_0
#define DRV_SPI_RX_DMA_THRESHOLD_IDX0    16 Driver Library
```

Configuring the Library

The configuration of the ENCx24J600 Driver Library is based on the file `sys_config.h`.

This header file contains the configuration selection for the ENCX24J600 Driver Library. Based on the selections made, the ENCx24J600 Driver Library may support the selected features. These configuration settings will apply to all instances of the ENCx24J600 Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

Building the Library

This section lists the files that are available in the ENCx24J600 Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/encx24j600`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source Folder Name	Description
/drv_encx24j600.h	This file provides the interface definitions of the ENCx24J600 Driver.

Required File(s)

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

Source Folder Name	Description
/src/dynamic/drv_drv_encx24J600_api.c	This file contains the API function implementations.
/src/dynamic/drv_encx24J600_main_state.c	This file contains the main state machine functions.
/src/dynamic/drv_encx24J600_utils.c	This file contains functions that are used throughout the driver.
/src/dynamic/bus/spi/drv_encx24J600_spi_bus.c	This file contains the functions to interface with the SPI bus.
/src/dynamic/closed_state/drv_encx24J600_closed_state.c	This file contains the functions for handling the driver closed state.
/src/dynamic/initialization_state/drv_encx24J600_configure_state.c	This file contains the functions for configuring the ENC hardware.
/src/dynamic/initialization_state/drv_encx24J600_detect_state.c	This file contains the functions for detecting the ENC hardware.
/src/dynamic/initialization_state/drv_encx24J600_initialization_state.c	This file contains the functions for the initialization state machine.

/src/dynamic/initialization_state/drv_encx24J600_reset_state.c	This file contains the functions for resetting the ENC hardware.
/src/dynamic/packet/drv_encx24J600_rx_packet.c	This file contains the functions for receiving a packet from the ENC hardware.
/src/dynamic/packet/drv_encx24J600_tx_packet.c	This file contains the functions for sending a packet to the ENC hardware.
/src/dynamic/running_state/drv_encx24J600_change_duplex_state.c	This file contains the functions for configuring the duplex mode of the ENC hardware.
/src/dynamic/running_state/drv_encx24J600_check_int_state.c	This file contains the functions for checking and processing the ENC hardware interrupts.
/src/dynamic/running_state/drv_encx24J600_check_status_state.c	This file contains the functions for checking the status of the ENC hardware.
/src/dynamic/running_state/drv_encx24J600_check_tx_status_state.c	This file contains the functions for checking the status of a transmitted packet.
/src/dynamic/running_state/drv_encx24J600_running_state.c	This file contains the functions for managing the running state machine.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source Folder Name	Description
N/A	No optional files exist for this library.

Module Dependencies

The ENCx24J600 Driver Library depends on the following modules:

- SPI Driver Library
- TCP/IP Stack Library

- TCP/IP Stack MAC Driver Module

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_ENCX24J600_Deinitialize	Deinitializes the ENCx24J600 Driver Instance. **Implementation:** Dynamic
	DRV_ENCX24J600_Initialize	Initializes the ENCx24J600 Driver Instance, with the configuration data. **Implementation:** Dynamic
	DRV_ENCX24J600_Reinitialize	Reinitializes the instance of the ENCX24J600 driver. **Implementation:** Dynamic
	DRV_ENCX24J600_Tasks	Main task function for the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_SetMacCtrlInfo	This function sets the MAC control information for the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_StackInitialize	This function initializes the driver with a TCPIP_MAC_INIT object. **Implementation:** Dynamic
	DRV_ENCX24J600_Process	Additional processing that happens outside the tasks function. **Implementation:** Dynamic

b) Client Level Functions

	Name	Description
	DRV_ENCX24J600_Close	Closes a client handle to the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_ConfigGet	Gets the current configuration. **Implementation:** Dynamic
	DRV_ENCX24J600_LinkCheck	This function returns the status of the link. **Implementation:** Dynamic
	DRV_ENCX24J600_Open	This function is called by the client to open a handle to a driver instance. **Implementation:** Dynamic
	DRV_ENCX24J600_ParametersGet	Get the parameters of the device. **Implementation:** Dynamic
	DRV_ENCX24J600_PowerMode	This function sets the power mode of the device. **Implementation:** Dynamic
	DRV_ENCX24J600_RegisterStatisticsGet	Get the register statistics. **Implementation:** Dynamic
	DRV_ENCX24J600_StatisticsGet	Retrieve the devices statistics. **Implementation:** Dynamic
	DRV_ENCX24J600_Status	Gets the current status of the driver. **Implementation:** Dynamic

c) Receive Functions

	Name	Description
	DRV_ENCX24J600_PacketRx	Receive a packet from the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_RxFilterHashTableEntrySet	This function adds an entry to the hash table. **Implementation:** Dynamic

d) Transmit Functions

	Name	Description
	DRV_ENCX24J600_PacketTx	This function queues a packet for transmission. **Implementation:** Dynamic

e) Event Functions

	Name	Description
	DRV_ENCX24J600_EventAcknowledge	Acknowledges an event. **Implementation:** Dynamic
	DRV_ENCX24J600_EventMaskSet	Sets the event mask. **Implementation:** Dynamic
	DRV_ENCX24J600_EventPendingGet	Gets the current events. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	_DRV_ENCX24J600_Configuration	Defines the data required to initialize or reinitialize the ENCX24J600 Driver.
	DRV_ENCX24J600_Configuration	Defines the data required to initialize or reinitialize the ENCX24J600 Driver.
	DRV_ENCX24J600_MDIX_TYPE	Defines the enumeration for controlling the MDIX select.

Description

This section describes the Application Programming Interface (API) functions of the ENCx24J600 Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_ENCX24J600_Deinitialize Function

Deinitializes the ENCx24J600 Driver Instance.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
void DRV_ENCX24J600_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

ENCX24J600 Deinitialization

This function deallocates any resources allocated by the initialization function.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize.

Parameters

Parameters	Description
Object	the valid object returned from DRV_ENCX24J600_Initialize

DRV_ENCX24J600_Initialize Function

Initializes the ENCx24J600 Driver Instance, with the configuration data.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
SYS_MODULE_OBJ DRV_ENCX24J600_Initialize(SYS_MODULE_INDEX index, SYS_MODULE_INIT * init);
```

Returns

- Valid handle to the driver instance - If successful
- SYS_MODULE_OBJ_INVALID - If unsuccessful

Description

ENCX24J600 Initialization

This function initializes the ENCx24J600 Driver with configuration data passed into it by either the system_init function or by the DRV_ENCX24J600_StackInitialize function. Calling this function alone is not enough to initialize the driver, DRV_ENCX24J600_SetMacCtrlInfo must be called with valid data before the driver is ready to be opened.

Preconditions

None.

Parameters

Parameters	Description
index	This is the index of the driver instance to be initialized. The definition DRV_ENCX24J600_NUM_DRV_INSTANCES controls how many instances are available.
init	This is a pointer to a DRV_ENX24J600_CONFIG structure.

DRV_ENCX24J600_Reinitialize Function

Reinitializes the instance of the ENCX24J600 driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
void DRV_ENCX24J600_Reinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

None

Description

ENCX24J600 Reinitialization

This function will deinitialize and initialize the driver instance. As with DRV_ENCX24J600_Initialize DRV_ENCX24J600_SetMacCtrlInfo must be called for the driver to be useful.

Remarks

This function is not planned to be implemented for the first release.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize.

DRV_ENCX24J600_Tasks Function

Main task function for the driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
void DRV_ENCX24J600_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

ENCX24J600 Tasks

This function will execute the main state machine for the ENCX24J600 driver.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize.

Parameters

Parameters	Description
object	The object valid passed back to DRV_ENCX24J600_Initialize

© 2013-2015 Microchip Technology Inc.

DRV_ENCX24J600_SetMacCtrlInfo Function

This function sets the MAC control information for the driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
void DRV_ENCX24J600_SetMacCtrlInfo(SYS_MODULE_OBJ object, TCPIP_MAC_MODULE_CTRL * init);
```

Returns

None.

Description

ENCX24J600 Set MAC Control Information

This function is used to pass in the TCPIP_MAC_CONTROL_INIT information that is used for allocation and deallocation of memory, event signaling, etc. This function is needed to be called so that the driver can enter initialization state when the tasks function is called.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize.

DRV_ENCX24J600_StackInitialize Function

This function initializes the driver with a TCPIP_MAC_INIT object.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
SYS_MODULE_OBJ DRV_ENCX24J600_StackInitialize(SYS_MODULE_INDEX index, const SYS_MODULE_INIT *
const init);
```

Returns

Returns a valid handle to the driver instance - If successful SYS_MODULE_OBJ_INVALID - If unsuccessful

Description

ENCX24J600 Stack Initialization

This function is used by the TCP/IP stack to fully initialize the driver with both the ENCX24J600 specific configuration and the MAC control information. With this function there is no need to call DRV_ENCX24J600_SetMacCtrlInfo.

Preconditions

None.

Parameters

Parameters	Description
index	This is the index of the driver instance to be initialized. The definition DRV_ENCX24J600_NUM_DRV_INSTANCES controls how many instances are available.
init	This is a pointer to a TCPIP_MAC_INIT structure.

DRV_ENCX24J600_Process Function

Additional processing that happens outside the tasks function.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_Process(DRV_HANDLE hMac);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENCX24J600 Process

This function does additional processing that is not done inside the tasks function.

Remarks

This function does nothing in the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

b) Client Level Functions

DRV_ENCX24J600_Close Function

Closes a client handle to the driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
void DRV_ENCX24J600_Close(DRV_HANDLE handle);
```

Returns

None.

Description

ENCX24J600 Close

This function closes a handle to the driver. If it is the last client open, the driver will send an RX Disable command to the ENC hardware and move to the closed state.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
handle	The successfully opened handle

DRV_ENCX24J600_ConfigGet Function

Gets the current configuration.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
size_t DRV_ENCX24J600_ConfigGet(DRV_HANDLE hMac, void* configBuff, size_t buffSize, size_t*
pConfigSize);
```

Returns

Number of bytes copied to the buffer

Description

ENCX24J600 Get Configuration

Gets the current configuration.

Remarks

This function does nothing in the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
configBuff	location to copy the configuration too
buffSize	buffer size
pConfigSize	configuration size needed

DRV_ENCX24J600_LinkCheck Function

This function returns the status of the link.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
bool DRV_ENCX24J600_LinkCheck(DRV_HANDLE hMac);
```

Returns

- true - if the link is active
- false - all other times

Description

ENCX24J600 Link Check

This function checks the status of the link and returns it to the caller.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

DRV_ENCX24J600_Open Function

This function is called by the client to open a handle to a driver instance.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
DRV_HANDLE DRV_ENCX24J600_Open(SYS_MODULE_INDEX index, DRV_IO_INTENT intent);
```

Returns

Returns a valid handle - If successful INVALID_HANDLE - If unsuccessful

Description

ENCX24J600 Open

The client will call this function to open a handle to the driver. When the first instance is opened than the driver will send the RX enabled command to the ENC hardware.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize.

Parameters

Parameters	Description
index	This is the index of the driver instance to be initialized. The definition DRV_ENCX24J600_NUM_DRV_INSTANCES controls how many instances are available.
intent	The intent to use when opening the driver. Only exclusive is supported

DRV_ENCX24J600_ParametersGet Function

Get the parameters of the device.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_ParametersGet(DRV_HANDLE hMac, TCPIP_MAC_PARAMETERS* pMacParams);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OK - if the hMac is valid

Description

ENCX24J600 Get Parameters

Get the parameters of the device, which includes that it is an Ethernet device and what it's MAC address is.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
pMacParams	pointer to put the parameters

DRV_ENCX24J600_PowerMode Function

This function sets the power mode of the device.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
bool DRV_ENCX24J600_PowerMode(DRV_HANDLE hMac, TCPIP_MAC_POWER_MODE pwrMode);
```

Returns

- false - This functionality is not supported in this version of the driver

Description

ENCX24J600 Power Mode

This function sets the power mode of the ENCX24J600.

Remarks

This functionality is not implemented in the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
pwrMode	the power mode to set

DRV_ENCX24J600_RegisterStatisticsGet Function

Get the register statistics.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_RegisterStatisticsGet(DRV_HANDLE hMac,
TCPIP_MAC_STATISTICS_REG_ENTRY* pRegEntries, int nEntries, int* pHwEntries);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENCX24J600 Get Register Statistics

Get the device specific statistics.

Remarks

Statistics are not planned for the first release

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
pRegEntries	
nEntries	
pHwEntries	

DRV_ENCX24J600_StatisticsGet Function

Retrieve the devices statistics.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_StatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_RX_STATISTICS*
pRxStatistics, TCPIP_MAC_TX_STATISTICS* pTxStatistics);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENCX24J600 Get Statistics

Get the current statistics stored in the driver.

Remarks

Statistics are not planned for the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

DRV_ENCX24J600_Status Function

Gets the current status of the driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
SYS_STATUS DRV_ENCX24J600_Status(SYS_MODULE_OBJ obect);
```

Returns

- SYS_STATUS_ERROR - if an invalid handle has been passed in
- SYS_STATUS_UNINITIALIZED - if the driver has not completed initialization
- SYS_STATUS_BUSY - if the driver is closing and moving to the closed state
- SYS_STATUS_READY - if the driver is ready for client commands

Description

ENCX24J600 Status

This function will get the status of the driver instance.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize().

Parameters

Parameters	Description
object	The object valid passed back to DRV_ENCX24J600_Initialize()

c) Receive Functions

DRV_ENCX24J600_PacketRx Function

Receive a packet from the driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_PACKET* DRV_ENCX24J600_PacketRx(DRV_HANDLE hMac, TCPIP_MAC_RES* pRes, const
TCPIP_MAC_PACKET_RX_STAT** ppPktStat);
```

Returns

- Pointer to a valid packet - if successful
- NULL - if unsuccessful

Description

ENCX24J600 Receive Packet

This function retrieves a packet from the driver. The packet needs to be acknowledged with the linked acknowledge function so it can be reused.

Remarks

ppPktStat is ignored in the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
pRes	the result of the operation
ppPktStat	pointer to the receive statistics

DRV_ENCX24J600_RxFilterHashTableEntrySet Function

This function adds an entry to the hash table.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_RxFilterHashTableEntrySet(DRV_HANDLE hMac, TCPIP_MAC_ADDR*
DestMACAddr);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENCX24J600 Receive Filter Hash Table Entry Set

This function adds to the MAC's hash table for hash table matching.

Remarks

This functionality is not implemented in the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
DestMACAddr	MAC address to add to the hash table

d) Transmit Functions

DRV_ENCX24J600_PacketTx Function

This function queues a packet for transmission.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_PacketTx(DRV_HANDLE hMac, TCPIP_MAC_PACKET * ptrPacket);
```

Returns

- TCPIP_MAC_RES_OP_ERR - if the client handle is invalid
- TCPIP_MAC_RES_IS_BUSY - if the driver is not in the run state
- TCPIP_MAC_RES_QUEUE_TX_FULL - if there are no free descriptors
- TCPIP_MAC_RES_OK - on successful queuing of the packet

Description

ENCX24J600 Packet Transmit

This function will take a packet and add it to the queue for transmission. When the packet has finished transmitting the driver will call the packets acknowledge function. When that acknowledge function is complete the driver will forget about the packet.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
ptrPacket	pointer to the packet

e) Event Functions

DRV_ENCX24J600_EventAcknowledge Function

Acknowledges an event.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
bool DRV_ENCX24J600_EventAcknowledge(DRV_HANDLE hMac, TCPIP_MAC_EVENT macEvents);
```

Returns

- true - if successful
- false - if not successful

Description

ENCX24J600 Acknowledge Event

This function acknowledges an event.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
macEvents	the events to acknowledge

DRV_ENCX24J600_EventMaskSet Function

Sets the event mask.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
bool DRV_ENCX24J600_EventMaskSet(DRV_HANDLE hMac, TCPIP_MAC_EVENT macEvents, bool enable);
```

Returns

- true - if the mask could be set
- false - if the mast could not be set

Description

ENCX24J600 Set Event Mask

Sets the event mask to what is passed in.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
macEvents	the mask to enable or disable
enable	to enable or disable events

DRV_ENCX24J600_EventPendingGet Function

Gets the current events.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_EVENT DRV_ENCX24J600_EventPendingGet(DRV_HANDLE hMac);
```

Returns

- TCPIP_MAC_EV_NONE - Returned on an error
- List of events - Returned on event other than an error

Description

ENCX24J600 Get Events

This function gets the current events.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

f) Data Types and Constants

DRV_ENCX24J600_Configuration Structure

Defines the data required to initialize or reinitialize the ENCX24J600 Driver.

File

drv_encx24j600.h

C

```
typedef struct _DRV_ENCX24J600_Configuration {
  uint16_t txDescriptors;
  uint16_t rxDescriptors;
  uint16_t rxDescBufferSize;
  SYS_MODULE_INDEX spiDrvIndex;
  uint32_t spiBps;
  uint16_t rxBufferSize;
  uint16_t maxFrameSize;
  PORTS_MODULE_ID spiSSPortModule;
  PORTS_CHANNEL spiSSPortChannel;
  PORTS_BIT_POS spiSSPortPin;
  bool intEnable;
  PORTS_MODULE_ID intPortModule;
  PORTS_CHANNEL intPortChannel;
  PORTS_BIT_POS intPortPin;
  DRV_ENCX24J600_MDIX_TYPE mdixControl;
  PORTS_MODULE_ID mdixPortModule;
  PORTS_CHANNEL mdixPortChannel;
  PORTS_BIT_POS mdixPortPin;
} DRV_ENCX24J600_Configuration;
```

Members

Members	Description
uint16_t txDescriptors;	Number of TX Descriptors to Allocate
uint16_t rxDescriptors;	Number of RX Descriptors to Allocate
uint16_t rxDescBufferSize;	Size of the buffer each RX Descriptor will use. Make sure its not smaller that maxFrameSize
SYS_MODULE_INDEX spiDrvIndex;	Index of the SPI driver to use
uint32_t spiBps;	Bus speed to use for the SPI interface. Section 1.0 of the ENCX24J600 data sheets says the maximum is 14000000 Hz. It is not recommended to go above this value.
uint16_t rxBufferSize;	The ENCX24J600 hardware has a 22 k dram. rxBufferSize defines how much of that memory is used by the rxBuffer
uint16_t maxFrameSize;	The maximum frame size to be supported by the hardware. 1536 is the default
PORTS_MODULE_ID spiSSPortModule;	Port Module of the GPIO pin hooked up to the CS/SS pin of the ENCX24J600
PORTS_CHANNEL spiSSPortChannel;	Port Channel of the GPIO pin hooked up to the CS/SS pin of the ENCX24J600
PORTS_BIT_POS spiSSPortPin;	Pin position of the GPIO pin hooked up to the CS/SS pin of the ENCX24J600
bool intEnable;	Use Interrupts or not.
PORTS_MODULE_ID intPortModule;	Port Module of the GPIO pin hooked up to the INT pin of the ENCX24J600
PORTS_CHANNEL intPortChannel;	Port Channel of the GPIO pin hooked up to the INT pin of the ENCX24J600
PORTS_BIT_POS intPortPin;	Pin Position of the GPIO pin hooked up to the INT pin of the ENCX24J600
DRV_ENCX24J600_MDIX_TYPE mdixControl;	To select the control type of the MDIX. This is only needed for hooking up to switches that don't have auto-mdix.
PORTS_MODULE_ID mdixPortModule;	Port Module of the GPIO pin hooked up to the MDIX select pin

PORTS_CHANNEL mdixPortChannel;	Port Channel of the GPIO pin hooked up to the MDIX select pin
PORTS_BIT_POS mdixPortPin;	Pin Position of the GPIO pin hooked up to the MDIX select pin

Description

ENCX24J600 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the ENCX24J600 driver. If the driver is built statically, the members of this data structure are statically over-ridden by static override definitions in the system_config.h file.

Remarks

None.

DRV_ENCX24J600_MDIX_TYPE Enumeration

Defines the enumeration for controlling the MDIX select.

File

drv_encx24j600.h

C

```
typedef enum {
  DRV_ENCX24J600_NO_CONTROL = 0,
  DRV_ENCX24J600_NORMAL,
  DRV_ENCX24J600_REVERSE = 0
} DRV_ENCX24J600_MDIX_TYPE;
```

Members

Members	Description
DRV_ENCX24J600_NO_CONTROL = 0	No Control
DRV_ENCX24J600_NORMAL	Normal MDIX
DRV_ENCX24J600_REVERSE = 0	Reverse MDIX

Description

ENCX24J600 Driver MDIX Control type

This type defines the enumeration for controlling the MDIX select.

Remarks

None.

Files

Files

Name	Description
drv_encx24j600.h	ENCx24J600 Driver interface definition.

Description

drv_encx24j600.h

ENCx24J600 Driver interface definition.

Enumerations

	Name	Description
	DRV_ENCX24J600_MDIX_TYPE	Defines the enumeration for controlling the MDIX select.

Functions

	Name	Description
	DRV_ENCX24J600_Close	Closes a client handle to the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_ConfigGet	Gets the current configuration. **Implementation:** Dynamic
	DRV_ENCX24J600_Deinitialize	Deinitializes the ENCx24J600 Driver Instance. **Implementation:** Dynamic
	DRV_ENCX24J600_EventAcknowledge	Acknowledges an event. **Implementation:** Dynamic
	DRV_ENCX24J600_EventMaskSet	Sets the event mask. **Implementation:** Dynamic
	DRV_ENCX24J600_EventPendingGet	Gets the current events. **Implementation:** Dynamic
	DRV_ENCX24J600_Initialize	Initializes the ENCx24J600 Driver Instance, with the configuration data. **Implementation:** Dynamic
	DRV_ENCX24J600_LinkCheck	This function returns the status of the link. **Implementation:** Dynamic
	DRV_ENCX24J600_Open	This function is called by the client to open a handle to a driver instance. **Implementation:** Dynamic
	DRV_ENCX24J600_PacketRx	Receive a packet from the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_PacketTx	This function queues a packet for transmission. **Implementation:** Dynamic
	DRV_ENCX24J600_ParametersGet	Get the parameters of the device. **Implementation:** Dynamic
	DRV_ENCX24J600_PowerMode	This function sets the power mode of the device. **Implementation:** Dynamic

	DRV_ENCX24J600_Process	Additional processing that happens outside the tasks function. **Implementation:** Dynamic
	DRV_ENCX24J600_RegisterStatisticsGet	Get the register statistics. **Implementation:** Dynamic
	DRV_ENCX24J600_Reinitialize	Reinitializes the instance of the ENCX24J600 driver. **Implementation:** Dynamic
	DRV_ENCX24J600_RxFilterHashTableEntrySet	This function adds an entry to the hash table. **Implementation:** Dynamic
	DRV_ENCX24J600_SetMacCtrlInfo	This function sets the MAC control information for the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_StackInitialize	This function initializes the driver with a TCPIP_MAC_INIT object. **Implementation:** Dynamic
	DRV_ENCX24J600_StatisticsGet	Retrieve the devices statistics. **Implementation:** Dynamic
	DRV_ENCX24J600_Status	Gets the current status of the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_Tasks	Main task function for the driver. **Implementation:** Dynamic

Structures

	Name	Description
	_DRV_ENCX24J600_Configuration	Defines the data required to initialize or reinitialize the ENCX24J600 Driver.
	DRV_ENCX24J600_Configuration	Defines the data required to initialize or reinitialize the ENCX24J600 Driver.

Description

ENCx24J600 Driver Public Interface

This file defines the interface definition for the ENCx24J600 Driver.

File Name

drv_encx24j600.h

Company

Microchip Technology Inc.

Ethernet MAC Driver Library

This topic describes the Ethernet MAC Driver Library.

Introduction

This library provides a driver-level abstraction of the on-chip Ethernet Controller found on many PIC32 devices. The driver implements the virtual MAC driver model that the MPLAB Harmony TCP/IP Stack requires. Please see the TCP/IP Stack Library MAC Driver Module help for details.

The "Host-To-Network" layer of a TCP/IP stack organization covers the Data Link and Physical Layers of the standard OSI stack. The Ethernet Controller provides the Data Link or Media Access Control Layer, in addition to other functions discussed in this section. An external Ethernet "PHY" provides the Physical layer, providing conversion between the digital and analog.

Description

The PIC32 Ethernet Controller is a bus master module that interfaces with an off-chip PHY to implement a complete Ethernet node in a system. The following are some of the key features of this module:

- Supports 10/100 Ethernet
  - Full-Duplex and Half-Duplex operation
  - Broadcast, Multicast and Unicast packets
  - Manual and automatic flow control
  - Supports Auto-MDIX enabled PHYs
  - Reduced Media Independent Interface (RMII) and Media Independent Interface (MII) PHY data interfaces
  - Performance statistics metrics in hardware.
- RAM descriptor based DMA operation for both receive and transmit path
- Fully configurable interrupts
- Configurable receive packet filtering using:
  - 64-bit Hash Table
  - 64-byte Pattern Match
  - Magic Packet™ Filtering
  - Runt Packet Detection and Filtering
- Supports Packet Payload Checksum calculation
- CRC Check

Support for the Serial Management Interface (SMI) (also known as the MIIM interface) is provided by the Ethernet PHY Driver Library.

Using the Library

The user of this driver is the MPLAB Harmony TCP/IP stack. This Ethernet driver is not intended as a system wide driver that the application or other system modules may use. It is intended for the sole use of the MPLAB Harmony TCP/IP stack and implements the virtual MAC model required by the stack.

This topic describes the basic architecture and functionality of the Ethernet MAC driver and is meant for advanced users or TCP/IP stack driver developers.

Interface Header File: `drv_ethmac.h`

The interface to the Ethernet MAC library is defined in the `drv_ethmac.h` header file, which is included by the MPLAB Harmony TCP/IP stack.

Please refer to the Understanding MPLAB Harmony section for how the library interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the Ethernet MAC Driver Library on Microchip's microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

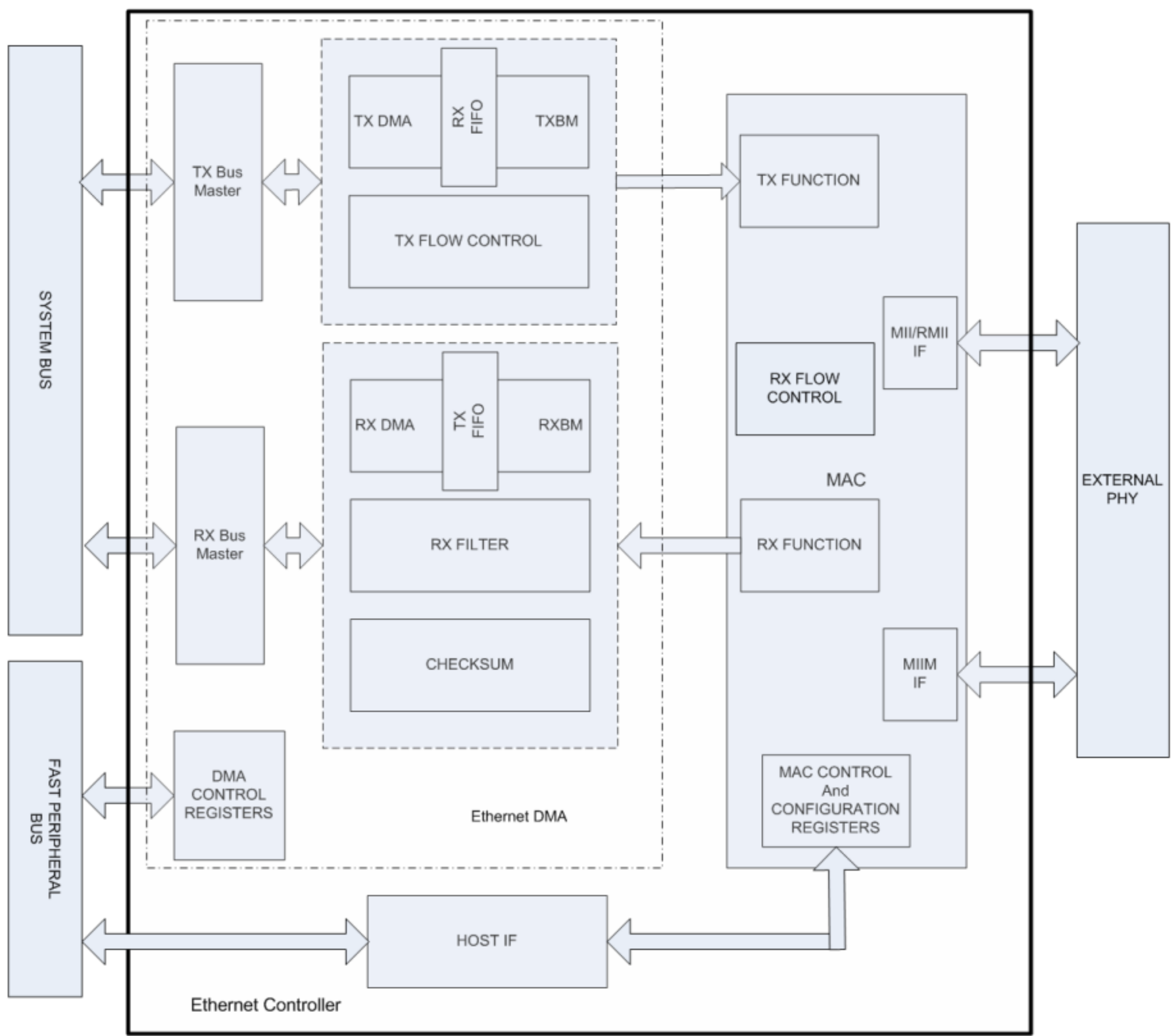
Description

The Ethernet Controller provides the modules needed to implement a 10/100 Mbps Ethernet node using an external Ethernet PHY chip. The PHY chip provides a digital-analog interface as part of the Physical Layer and the controller provides the Media Access Controller (MAC) layer above the PHY.

As shown in Figure 1, the Ethernet Controller consists of the following modules:

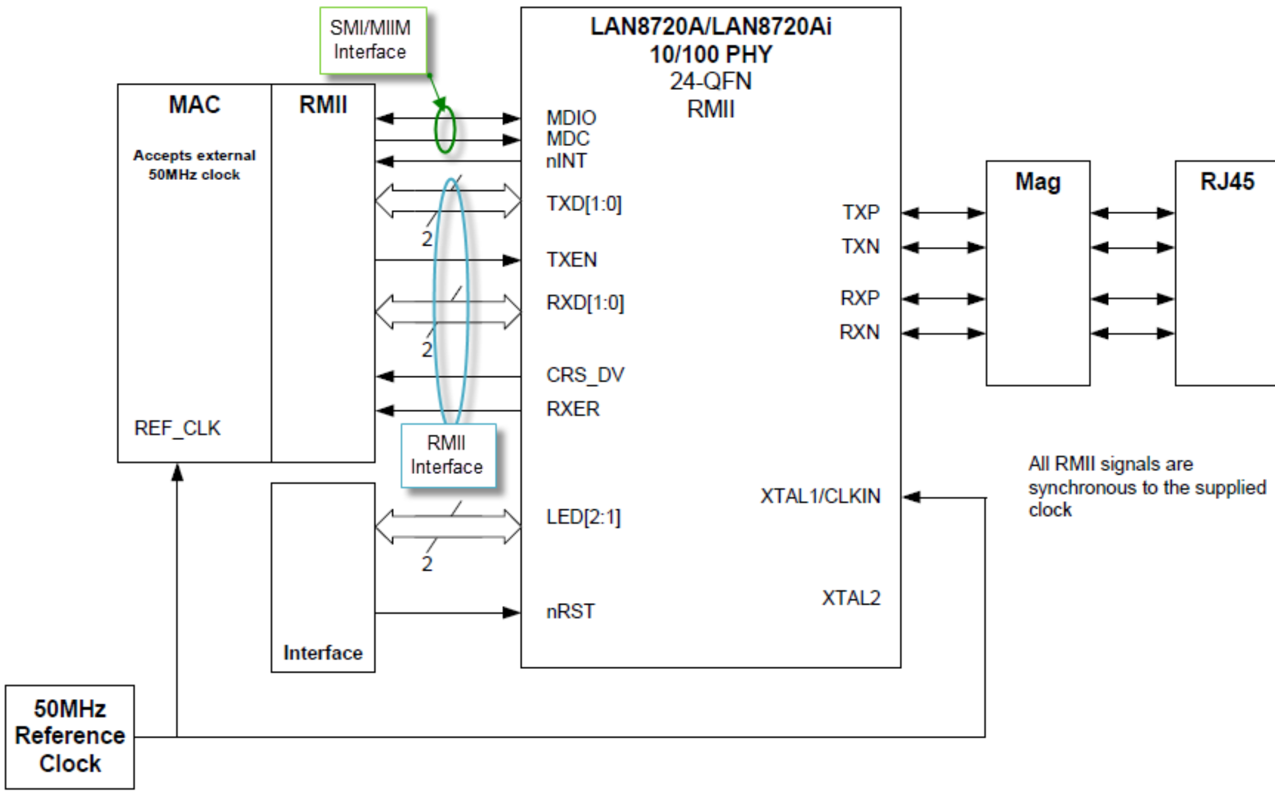
- Media Access Control (MAC) block: Responsible for implementing the MAC functions of the Ethernet IEEE 802.3 Specification
- Flow Control (FC) block: Responsible for control of the transmission of PAUSE frames. (Reception of PAUSE frames is handled within the MAC.)
- RX Filter (RXF) block: This module performs filtering on every receive packet to determine whether each packet should be accepted or rejected
- TX DMA/TX Buffer Management Engine: The TX DMA and TX Buffer Management engines perform data transfers from the memory (using descriptor tables) to the MAC Transmit Interface
- RX DMA/RX Buffer Management Engine: The RX DMA and RX Buffer Management engines transfer receive packets from the MAC to the memory (using descriptor tables)

Figure 1: Ethernet Controller Block Diagram



For completeness, we also need to look at the interface diagram of a representative Ethernet PHY. As shown in Figure 2, the PHY has two interfaces, one for configuring and managing the PHY (SMI/MIIM) and another for transmit and receive data (RMII or MII). The SMI/MIIM interface is the responsibility of the Ethernet PHY Driver Library. When setting up the Ethernet PHY, this Ethernet driver calls primitives from the Ethernet PHY Driver library. The RMII/MII data interface is the responsibility of the Ethernet MAC Driver Library (this library).

Figure 2: Ethernet PHY Interfaces



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system. Refer to the TCP/IP Stack Library MAC Driver Module help for the interface that the Ethernet driver has to implement in a MPLAB Harmony system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Ethernet MAC Driver Library.

Library Interface Section	Description
Client Level Functions	DRV_ETHMAC_PIC32MACOpen, DRV_ETHMAC_PIC32MACClose, and DRV_ETHMAC_PIC32MACSetup to support the TCP/IP Stack. Plus link status and power options.
Receive Functions	Receive routines.
Transmit Functions	Transmit routines.
Event Functions	Ethernet event support routines.
Other Functions	Additional routines.
Data Types and Constants	Typedefs and #defines.

Configuring the Library

Macros

	Name	Description
	DRV_ETHMAC_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ETHMAC_INDEX	Ethernet MAC static index selection.
	DRV_ETHMAC_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_ETHMAC_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_ETHMAC_INTERRUPT_SOURCE	Defines an override of the interrupt source in case of static driver.
	DRV_ETHMAC_PERIPHERAL_ID	Defines an override of the peripheral ID.
	DRV_ETHMAC_POWER_STATE	Defines an override of the power state of the Ethernet MAC driver.

Description

The configuration of the Ethernet MAC driver is done as part of the MPLAB Harmony TCP/IP Stack configuration and is based on the `system_config.h` file, which may include the `tcpip_mac_config.h`. See the TCP/IP Stack Library MAC Driver Module help file for configuration options.

This header file contains the configuration selection for the Ethernet MAC Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

DRV_ETHMAC_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_CLIENTS_NUMBER 1
```

Description

Ethernet MAC Maximum Number of Clients

This definition select the maximum number of clients that the Ethernet MAC driver can support at run time. Not defining it means using a single client.

Remarks

None.

DRV_ETHMAC_INDEX Macro

Ethernet MAC static index selection.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_INDEX DRV_ETHMAC_INDEX_1
```

Description

Ethernet MAC Static Index Selection

This definition selects the Ethernet MAC static index for the driver object reference

Remarks

This index is required to make a reference to the driver object.

DRV_ETHMAC_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_INSTANCES_NUMBER 1
```

Description

Ethernet MAC hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver. Not defining it means using a static driver.

Remarks

None.

DRV_ETHMAC_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_INTERRUPT_MODE true
```

Description

Ethernet MAC Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of timer operation is desired
- false - Select if polling mode of timer operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

DRV_ETHMAC_INTERRUPT_SOURCE Macro

Defines an override of the interrupt source in case of static driver.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_INTERRUPT_SOURCE INT_SOURCE_ETH_1
```

Description

Ethernet MAC Interrupt Source

Defines an override of the interrupt source in case of static driver.

Remarks

Refer to the INT PLIB document for more information on INT_SOURCE enumeration.

DRV_ETHMAC_PERIPHERAL_ID Macro

Defines an override of the peripheral ID.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_PERIPHERAL_ID ETHMAC_ID_1
```

Description

Ethernet MAC Peripheral ID Selection

Defines an override of the peripheral ID, using macros.

Remarks

Some devices also support ETHMAC_ID_0

DRV_ETHMAC_POWER_STATE Macro

Defines an override of the power state of the Ethernet MAC driver.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_POWER_STATE SYS_MODULE_POWER_IDLE_STOP
```

Description

Ethernet MAC power state configuration

Defines an override of the power state of the Ethernet MAC driver.

Remarks

This feature may not be available in the device or the Ethernet MAC module selected.

Building the Library

This section lists the files that are available in the Ethernet MAC Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/ethmac`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_ethmac.h	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_ethmac.c	PIC32 internal Ethernet driver virtual MAC implementation file.
/src/dynamic/drv_ethmac_lib.c	PIC32 internal Ethernet driver controller implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The Ethernet MAC Driver Library depends on the following modules:

- Ethernet PHY Driver Library
- Interrupt System Service Library
- Timer System Service Library
- Ethernet Peripheral Library

Library Interface

a) Client Level Functions

	Name	Description
	DRV_ETHMAC_PIC32MACClose	Closes a client instance of the PIC32 MAC Driver. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACDeinitialize	Deinitializes the PIC32 Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACInitialize	Initializes the PIC32 Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACLinkCheck	Checks current link status. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACOpen	Opens a client instance of the PIC32 MAC Driver. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACParametersGet	MAC parameter get function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACPowerMode	Selects the current power mode for the Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACProcess	MAC periodic processing function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACStatisticsGet	Gets the current MAC statistics. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACStatus	Provides the current status of the MAC driver module. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACConfigGet	Gets the current MAC driver configuration. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACRegisterStatisticsGet	Gets the current MAC hardware statistics registers. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACReinitialize	Reinitializes the PIC32 Ethernet MAC. **Implementation:** Dynamic

b) Receive Functions

	Name	Description
	DRV_ETHMAC_PIC32MACPacketRx	This is the MAC receive function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet	Sets the current MAC hash table receive filter. **Implementation:** Dynamic

c) Transmit Functions

	Name	Description
	DRV_ETHMAC_PIC32MACPacketTx	MAC driver transmit function. **Implementation:** Dynamic

d) Event Functions

	Name	Description
	DRV_ETHMAC_PIC32MACEventAcknowledge	Acknowledges and re-enables processed events. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACEventMaskSet	Enables/disables the MAC events. **Implementation:** Dynamic

	DRV_ETHMAC_PIC32MACEventPendingGet	Returns the currently pending events. **Implementation:** Dynamic

e) Other Functions

	Name	Description
	DRV_ETHMAC_Tasks_ISR	Ethernet MAC driver interrupt function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACTasks	Maintains the EThernet MAC driver's state machine. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	DRV_ETHMAC_INDEX_1	This is macro DRV_ETHMAC_INDEX_1.
	DRV_ETHMAC_INDEX_0	Ethernet driver index definitions.
	DRV_ETHMAC_INDEX_COUNT	Number of valid Ethernet driver indices.

Description

This section lists the interface routines, data types, constants and macros for the library.

a) Client Level Functions

DRV_ETHMAC_PIC32MACClose Function

Closes a client instance of the PIC32 MAC Driver.

Implementation: Dynamic

File

drv_ethmac.h

C

```
void DRV_ETHMAC_PIC32MACClose(DRV_HANDLE hMac);
```

Returns

None

Description

This function closes a client instance of the PIC32 MAC Driver.

Remarks

None

Preconditions

DRV_ETHMAC_PIC32MACOpen() should have been called.

Example

Parameters

Parameters	Description
hMac	valid MAC handle, obtained by a call to DRV_ETHMAC_PIC32MACOpen

Function

void DRV_ETHMAC_PIC32MACClose(DRV_HANDLE hMac)

DRV_ETHMAC_PIC32MACDeinitialize Function

Deinitializes the PIC32 Ethernet MAC.

Implementation: Dynamic

File

drv_ethmac.h

C

```
void DRV_ETHMAC_PIC32MACDeinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function supports teardown of the PIC32 Ethernet MAC (opposite of set up). Used by tcpip_module_manager.

Remarks

This function deinitializes the Ethernet controller, the MAC and the associated PHY. It should be called to be release any resources allocated by the initialization and return the MAC and the PHY to the idle/power down state.

Preconditions

DRV_ETHMAC_PIC32MACInitialize must have been called to set up the driver.

Example

Function

void DRV_ETHMAC_PIC32MACDeinitialize(SYS_MODULE_OBJ object);

DRV_ETHMAC_PIC32MACInitialize Function

Initializes the PIC32 Ethernet MAC.

Implementation: Dynamic

File

drv_ethmac.h

C

```
SYS_MODULE_OBJ DRV_ETHMAC_PIC32MACInitialize(const SYS_MODULE_INDEX index, const
SYS_MODULE_INIT * const init);
```

Returns

- a valid handle to a driver object, if successful.
- SYS_MODULE_OBJ_INVALID if initialization failed.

Description

This function supports the initialization of the PIC32 Ethernet MAC. Used by tcpip_module_manager.

Remarks

This function initializes the Ethernet controller, the MAC and the associated PHY. It should be called to be able to schedule any Ethernet transmit or receive operation.

Preconditions

None

Example

Function

SYS_MODULE_OBJ DRV_ETHMAC_PIC32MACInitialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);

DRV_ETHMAC_PIC32MACLinkCheck Function

Checks current link status.

Implementation: Dynamic

File

drv_ethmac.h

C

```
bool DRV_ETHMAC_PIC32MACLinkCheck(DRV_HANDLE hMac);
```

Returns

- true - If the link is up
- false - If the link is not up

Description

This function checks the link status of the associated network interface.

Remarks

The function will automatically perform a MAC reconfiguration if the link went up after being down and the PHY auto negotiation is enabled.

Preconditions

DRV_ETHMAC_PIC32MACInitialize must have been called to set up the driver. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Parameters

Parameters	Description
hMac	Ethernet MAC client handle

Function

bool DRV_ETHMAC_PIC32MACLinkCheck(DRV_HANDLE hMac)

DRV_ETHMAC_PIC32MACOpen Function

Opens a client instance of the PIC32 MAC Driver.

Implementation: Dynamic

File

drv_ethmac.h

C

```
DRV_HANDLE DRV_ETHMAC_PIC32MACOpen(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

- DRV_HANDLE - handle (pointer) to MAC client
- 0 if call failed

Description

This function opens a client instance of the PIC32 MAC Driver. Used by tcpip_module_manager.

Remarks

The intent parameter is not used in the current implementation and is maintained only for compatibility with the generic driver Open function signature.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called.

Example

Function

DRV_HANDLE DRV_ETHMAC_PIC32MACOpen(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);

DRV_ETHMAC_PIC32MACParametersGet Function

MAC parameter get function.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACParametersGet(DRV_HANDLE hMac, TCPIP_MAC_PARAMETERS*
pMacParams);
```

Returns

- TCPIP_MAC_RES_OK if pMacParams updated properly
- a TCPIP_MAC_RES error code if processing failed for some reason

Description

MAC Parameter Get function TCPIP_MAC_RES DRV_ETHMAC_PIC32MACParametersGet(DRV_HANDLE hMac, TCPIP_MAC_PARAMETERS* pMacParams);

This is a function that returns the run time parameters of the MAC driver.

Remarks

None.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

DRV_ETHMAC_PIC32MACPowerMode Function

Selects the current power mode for the Ethernet MAC.

Implementation: Dynamic

File

drv_ethmac.h

C

```
bool DRV_ETHMAC_PIC32MACPowerMode(DRV_HANDLE hMac, TCPIP_MAC_POWER_MODE pwrMode);
```

Returns

- true if the call succeeded.
- false if the call failed

Description

This function sets the power mode for the Ethernet MAC.

Remarks

This function is not currently supported by the Ethernet MAC and will always return true.

Preconditions

DRV_ETHMAC_PIC32MACInitialize must have been called to set up the driver. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

bool DRV_ETHMAC_PIC32MACPowerMode(DRV_HANDLE hMac, TCPIP_MAC_POWER_MODE pwrMode)

DRV_ETHMAC_PIC32MACProcess Function

MAC periodic processing function.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACProcess(DRV_HANDLE hMac);
```

Returns

- TCPIP_MAC_RES_OK if all processing went on OK
- a TCPIP_MAC_RES error code if processing failed for some reason

Description

This is a function that allows for internal processing by the MAC driver. It is meant for processing that cannot be done from within ISR.

Normally this function will be called in response to an TX and/or RX event signaled by the driver. This is specified by the MAC driver at initialization time using TCPIP_MAC_MODULE_CTRL.

Remarks

- The MAC driver may use the DRV_ETHMAC_PIC32MACProcess() for:
  - Processing its pending TX queues
  - RX buffers replenishing functionality. If the number of packets in the RX queue falls below a specified limit, the MAC driver may use this function to allocate some extra RX packets. Similarly, if there are too many allocated RX packets, the MAC driver can free some of them.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Parameters

Parameters	Description
hMac	Ethernet MAC client handle

Function

TCPIP_MAC_RES DRV_ETHMAC_PIC32MACProcess(DRV_HANDLE hMac);

DRV_ETHMAC_PIC32MACStatisticsGet Function

Gets the current MAC statistics.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACStatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_RX_STATISTICS*
pRxStatistics, TCPIP_MAC_TX_STATISTICS* pTxStatistics);
```

Returns

- TCPIP_MAC_RES_OK if all processing went on OK.
- TCPIP_MAC_RES_OP_ERR error code if function not supported by the driver.

Description

This function will get the current value of the statistic counters maintained by the MAC driver.

Remarks

- The reported values are info only and change dynamically.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

TCPIP_MAC_RES DRV_ETHMAC_PIC32MACStatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_RX_STATISTICS* pRxStatistics, TCPIP_MAC_TX_STATISTICS* pTxStatistics);

DRV_ETHMAC_PIC32MACStatus Function

Provides the current status of the MAC driver module.

Implementation: Dynamic

File

drv_ethmac.h

C

```
SYS_STATUS DRV_ETHMAC_PIC32MACStatus(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed
- SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed
- SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This function provides the current status of the MAC driver module.

Remarks

None.

Preconditions

The DRV_ETHMAC_PIC32MACInitialize function must have been called before calling this function.

Example

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_ETHMAC_PIC32MACInitialize

Function

SYS_STATUS DRV_ETHMAC_PIC32MACStatus (SYS_MODULE_OBJ object)

DRV_ETHMAC_PIC32MACConfigGet Function

Gets the current MAC driver configuration.

Implementation: Dynamic

File

drv_ethmac.h

C

```
size_t DRV_ETHMAC_PIC32MACConfigGet(DRV_HANDLE hMac, void* configBuff, size_t buffSize, size_t*
pConfigSize);
```

Returns

- number of bytes copied into the supplied storage buffer

Description

This function will get the current MAC driver configuration and store it into a supplied buffer.

Remarks

- None

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

size_t DRV_ETHMAC_PIC32MACConfigGet(DRV_HANDLE hMac, void* configBuff, size_t buffSize, size_t* pConfigSize);

DRV_ETHMAC_PIC32MACRegisterStatisticsGet Function

Gets the current MAC hardware statistics registers.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACRegisterStatisticsGet(DRV_HANDLE hMac,
TCPIP_MAC_STATISTICS_REG_ENTRY* pRegEntries, int nEntries, int* pHwEntries);
```

Returns

- TCPIP_MAC_RES_OK if all processing went on OK.
- TCPIP_MAC_RES_OP_ERR error code if function not supported by the driver.

Description

This function will get the current value of the statistic registers of the associated MAC controller.

Remarks

- The reported values are info only and change dynamically.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

TCPIP_MAC_RES DRV_ETHMAC_PIC32MACRegisterStatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_STATISTICS_REG_ENTRY* pRegEntries, int nEntries, int* pHwEntries);

DRV_ETHMAC_PIC32MACReinitialize Function

Reinitializes the PIC32 Ethernet MAC.

Implementation: Dynamic

File

drv_ethmac.h

C

```
void DRV_ETHMAC_PIC32MACReinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

None.

Description

This function supports re-initialization of the PIC32 Ethernet MAC (opposite of set up).

Remarks

This function is not supported yet.

Preconditions

DRV_ETHMAC_PIC32MACInitialize must have been called to set up the driver.

Example

Function

void DRV_ETHMAC_PIC32MACReinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);

b) Receive Functions

DRV_ETHMAC_PIC32MACPacketRx Function

This is the MAC receive function.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_PACKET* DRV_ETHMAC_PIC32MACPacketRx(DRV_HANDLE hMac, TCPIP_MAC_RES* pRes, const
TCPIP_MAC_PACKET_RX_STAT** ppPktStat);
```

Returns

- a valid pointer to an available RX packet
- 0 if no packet pending/available

Description

This function will return a packet if such a pending packet exists.

Additional information about the packet is available by providing the pRes and ppPktStat fields.

Remarks

- Once a pending packet is available in the MAC driver internal RX queues this function will dequeue the packet and hand it over to the MAC driver's client - i.e., the stack - for further processing.

- The flags for a RX packet are updated by the MAC driver:
  - TCPIP_MAC_PKT_FLAG_RX will be set
  - TCPIP_MAC_PKT_FLAG_UNICAST is set if that packet is a unicast packet
  - TCPIP_MAC_PKT_FLAG_BCAST is set if that packet is a broadcast packet
  - TCPIP_MAC_PKT_FLAG_MCAST is set if that packet is a multicast packet
  - TCPIP_MAC_PKT_FLAG_QUEUED is set
  - TCPIP_MAC_PKT_FLAG_SPLIT is set if the packet has multiple data segments

- The MAC driver dequeues and return to the caller just one single packet. That is the packets are not chained.

- The packet buffers are allocated by the Ethernet MAC driver itself, Once the higher level layers in the stack are done with processing the RX packet, they have to call the corresponding packet acknowledgment function that tells the MAC driver that it can resume control of that packet.

- Once the stack modules are done processing the RX packets and the acknowledge function is called the MAC driver will reuse the RX packets.

- The MAC driver may use the DRV_ETHMAC_PIC32MACProcess() for obtaining new RX packets if needed.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

TCPIP_MAC_PACKET* DRV_ETHMAC_PIC32MACPacketRx (DRV_HANDLE hMac, TCPIP_MAC_RES* pRes, const TCPIP_MAC_PACKET_RX_STAT** ppPktStat);

DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet Function

Sets the current MAC hash table receive filter.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet(DRV_HANDLE hMac, TCPIP_MAC_ADDR*
DestMACAddr);
```

Returns

- TCPIP_MAC_RES_OK if success
- a TCPIP_MAC_RES error value if failed

Description

This function sets the MAC hash table filtering to allow packets sent to DestMACAddr to be received. It calculates a CRC-32 using polynomial 0x4C11DB7 over the 6 byte MAC address and then, using bits 28:23 of the CRC, will set the appropriate bits in the hash table filter registers (ETHHT0-ETHHT1).

The function will enable/disable the Hash Table receive filter if needed.

Remarks

- Sets the appropriate bit in the ETHHT0/1 registers to allow packets sent to DestMACAddr to be received and enabled the Hash Table receive filter.
- There is no way to individually remove destination MAC addresses from the hash table since it is possible to have a hash collision and therefore multiple MAC addresses relying on the same hash table bit.
- A workaround is to have the stack store each enabled MAC address and to perform the comparison at run time.
- A call to DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet() using a 00-00-00-00-00-00 destination MAC address, which will clear the entire hash table and disable the hash table filter. This will allow the receive of all packets, regardless of their destination

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

TCPIP_MAC_RES DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet(DRV_HANDLE hMac, TCPIP_MAC_ADDR* DestMACAddr)

c) Transmit Functions

DRV_ETHMAC_PIC32MACPacketTx Function

MAC driver transmit function.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACPacketTx(DRV_HANDLE hMac, TCPIP_MAC_PACKET * ptrPacket);
```

Returns

- TCPIP_MAC_RES_OK if success
- a TCPIP_MAC_RES error value if failed

Description

This is the MAC transmit function. Using this function a packet is submitted to the MAC driver for transmission.

Remarks

- The MAC driver supports internal queuing. A packet is rejected only if it's not properly formatted. Otherwise it will be scheduled for transmission and queued internally if needed.

- Once the packet is scheduled for transmission the MAC driver will set the TCPIP_MAC_PKT_FLAG_QUEUED flag so that the stack is aware that this packet is under processing and cannot be modified.

- Once the packet is transmitted, the TCPIP_MAC_PKT_FLAG_QUEUED will be cleared, the proper packet acknowledgment result (ackRes) will be set and the packet acknowledgment function (ackFunc) will be called.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

TCPIP_MAC_RES DRV_ETHMAC_PIC32MACPacketTx(DRV_HANDLE hMac, TCPIP_MAC_PACKET * ptrPacket);

d) Event Functions

DRV_ETHMAC_PIC32MACEventAcknowledge Function

Acknowledges and re-enables processed events.

Implementation: Dynamic

File

drv_ethmac.h

C

```
bool DRV_ETHMAC_PIC32MACEventAcknowledge(DRV_HANDLE hMac, TCPIP_MAC_EVENT tcpAckEv);
```

Returns

- true if events acknowledged
- false if no events to be acknowledged

Description

This function acknowledges and re-enables processed events. Multiple events can be ORed together as they are processed together. The events acknowledged by this function should be the events that have been retrieved from the stack by calling DRV_ETHMAC_PIC32MACEventPendingGet() or have been passed to the stack by the driver using the registered notification handler and have been processed and have to be re-enabled.

Remarks

- All events should be acknowledged, in order to be re-enabled.

- Some events are fatal errors and should not be acknowledged (TCPIP_MAC_EV_RX_BUSERR, TCPIP_MAC_EV_TX_BUSERR). Driver/stack re-initialization is needed under such circumstances.

- Some events are just system/application behavior and they are intended only as simple info (TCPIP_MAC_EV_RX_OVFLOW, TCPIP_MAC_EV_RX_BUFNA, TCPIP_MAC_EV_TX_ABORT, TCPIP_MAC_EV_RX_ACT).

- The TCPIP_MAC_EV_RX_FWMARK and TCPIP_MAC_EV_RX_EWMARK events are part of the normal flow control operation (if auto flow control was enabled). They should be enabled alternatively, if needed.

- The events are persistent. They shouldn't be re-enabled unless they have been processed and the condition that generated them was removed. Re-enabling them immediately without proper processing will have dramatic effects on system performance.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

```
DRV_ETHMAC_PIC32MACEventAcknowledge( hMac, stackNewEvents );
```

Function

bool DRV_ETHMAC_PIC32MACEventAcknowledge(DRV_HANDLE hMac, TCPIP_MAC_EVENT tcpAckEv);

DRV_ETHMAC_PIC32MACEventMaskSet Function

Enables/disables the MAC events.

Implementation: Dynamic

File

drv_ethmac.h

C

```
bool DRV_ETHMAC_PIC32MACEventMaskSet(DRV_HANDLE hMac, TCPIP_MAC_EVENT macEvents, bool enable);
```

Returns

always true, operation succeeded.

Description

This is a function that enables or disables the events to be reported to the Ethernet MAC client (TCP/IP stack).

All events that are to be enabled will be added to the notification process. All events that are to be disabled will be removed from the notification process. The stack has to catch the events that are notified and process them. After that the stack should call DRV_ETHMAC_PIC32MACEventAcknowledge() so that the events can be re-enable

The stack should process at least the following transfer events:

- TCPIP_MAC_EV_RX_PKTPEND
- TCPIP_MAC_EV_RX_DONE
- TCPIP_MAC_EV_TX_DONE

Remarks

- The event notification system enables the user of the TCP/IP stack to call into the stack for processing only when there are relevant events rather than being forced to periodically call from within a loop.

- If the notification events are nil, the interrupt processing will be disabled. Otherwise, the event notification will be enabled and the interrupts relating to the requested events will be enabled.

- Note that once an event has been caught by the stack ISR (and reported if a notification handler is in place) it will be disabled until the DRV_ETHMAC_PIC32MACEventAcknowledge() is called.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

```
DRV_ETHMAC_PIC32MACEventMaskSet( hMac, TCPIP_MAC_EV_RX_OVFLOW | TCPIP_MAC_EV_RX_BUFNA, true );
```

Function

bool DRV_ETHMAC_PIC32MACEventMaskSet(DRV_HANDLE hMac, TCPIP_MAC_EVENT macEvents, bool enable);

DRV_ETHMAC_PIC32MACEventPendingGet Function

Returns the currently pending events.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_EVENT DRV_ETHMAC_PIC32MACEventPendingGet(DRV_HANDLE hMac);
```

Returns

The currently stack pending events.

Description

This function returns the currently pending Ethernet MAC events. Multiple events will be ORed together as they accumulate. The stack should perform processing whenever a transmission related event (TCPIP_MAC_EV_RX_PKTPEND, TCPIP_MAC_EV_TX_DONE) is present. The other, non critical events, may not be managed by the stack and passed to an user. They will have to be eventually acknowledged if re-enabling is needed.

Remarks

- This is the preferred method to get the current pending MAC events. The stack maintains a proper image of the events from their occurrence to their acknowledgment.
- Even with a notification handler in place it's better to use this function to get the current pending events rather than using the events passed by the notification handler which could be stale.
- The events are persistent. They shouldn't be re-enabled unless they have been processed and the condition that generated them was removed. Re-enabling them immediately without proper processing will have dramatic effects on system performance.
- The returned value is just a momentary value. The pending events can change any time.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

```
TCPIP_MAC_EVENT currEvents = DRV_ETHMAC_PIC32MACEventPendingGet( hMac);
```

Function

TCPIP_MAC_EVENT DRV_ETHMAC_PIC32MACEventPendingGet(DRV_HANDLE hMac)

e) Other Functions

DRV_ETHMAC_Tasks_ISR Function

Ethernet MAC driver interrupt function.

Implementation: Dynamic

File

drv_ethmac.h

C

```
void DRV_ETHMAC_Tasks_ISR(SYS_MODULE_OBJ macIndex);
```

Returns

None.

Description

This is the Ethernet MAC driver interrupt service routine. It processes the Ethernet related interrupts and notifies the events to the driver user (the TCP/IP stack).

Remarks

None.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. The TCP/IP stack event notification should be enabled.

Function

void DRV_ETHMAC_Tasks_ISR(SYS_MODULE_OBJ macIndex)

DRV_ETHMAC_PIC32MACTasks Function

Maintains the EThernet MAC driver's state machine.

Implementation: Dynamic

File

drv_ethmac.h

C

```
void DRV_ETHMAC_PIC32MACTasks(SYS_MODULE_OBJ object);
```

Returns

None

Description

This function is used to maintain the driver's internal state machine

Remarks

None.

Preconditions

The DRV_ETHMAC_PIC32MACInitialize routine must have been called for the specified MAC driver instance.

Example

Function

void DRV_ETHMAC_PIC32MACTasks(SYS_MODULE_OBJ object)

f) Data Types and Constants

DRV_ETHMAC_INDEX_1 Macro

File

drv_ethmac.h

C

```
#define DRV_ETHMAC_INDEX_1 1
```

Description

This is macro DRV_ETHMAC_INDEX_1.

DRV_ETHMAC_INDEX_0 Macro

Ethernet driver index definitions.

File

drv_ethmac.h

C

```
#define DRV_ETHMAC_INDEX_0 0
```

Description

Ethernet Driver Module Index Numbers

These constants provide Ethernet driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the MAC initialization routines to identify the driver instance in use.

DRV_ETHMAC_INDEX_COUNT Macro

Number of valid Ethernet driver indices.

File

drv_ethmac.h

C

```
#define DRV_ETHMAC_INDEX_COUNT ETH_NUMBER_OF_MODULES
```

Description

Ethernet Driver Module Index Count

This constant identifies number of valid Ethernet driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from part-specific header files defined as part of the peripheral libraries.

Files

Files

Name	Description
drv_ethmac.h	Ethernet MAC device driver interface file
drv_ethmac_config.h	Ethernet MAC driver configuration definitions template.

Description

This section lists the source and header files used by the Ethernet MAC Driver Library.

drv_ethmac.h

Ethernet MAC device driver interface file

Functions

	Name	Description
	DRV_ETHMAC_PIC32MACClose	Closes a client instance of the PIC32 MAC Driver. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACConfigGet	Gets the current MAC driver configuration. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACDeinitialize	Deinitializes the PIC32 Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACEventAcknowledge	Acknowledges and re-enables processed events. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACEventMaskSet	Enables/disables the MAC events. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACEventPendingGet	Returns the currently pending events. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACInitialize	Initializes the PIC32 Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACLinkCheck	Checks current link status. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACOpen	Opens a client instance of the PIC32 MAC Driver. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACPacketRx	This is the MAC receive function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACPacketTx	MAC driver transmit function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACParametersGet	MAC parameter get function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACPowerMode	Selects the current power mode for the Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACProcess	MAC periodic processing function. **Implementation:** Dynamic

	DRV_ETHMAC_PIC32MACRegisterStatisticsGet	Gets the current MAC hardware statistics registers. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACReinitialize	Reinitializes the PIC32 Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet	Sets the current MAC hash table receive filter. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACStatisticsGet	Gets the current MAC statistics. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACStatus	Provides the current status of the MAC driver module. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACTasks	Maintains the EThernet MAC driver's state machine. **Implementation:** Dynamic
	DRV_ETHMAC_Tasks_ISR	Ethernet MAC driver interrupt function. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_ETHMAC_INDEX_0	Ethernet driver index definitions.
	DRV_ETHMAC_INDEX_1	This is macro DRV_ETHMAC_INDEX_1.
	DRV_ETHMAC_INDEX_COUNT	Number of valid Ethernet driver indices.

Description

Ethernet MAC Device Driver Interface

The Ethernet MAC device driver provides a simple interface to manage the Ethernet peripheral. This file defines the interface definitions and prototypes for the Ethernet MAC driver.

File Name

drv_ethmac.h

Company

Microchip Technology Inc.

drv_ethmac_config.h

Ethernet MAC driver configuration definitions template.

Macros

	Name	Description
	DRV_ETHMAC_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ETHMAC_INDEX	Ethernet MAC static index selection.
	DRV_ETHMAC_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_ETHMAC_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_ETHMAC_INTERRUPT_SOURCE	Defines an override of the interrupt source in case of static driver.
	DRV_ETHMAC_PERIPHERAL_ID	Defines an override of the peripheral ID.

	DRV_ETHMAC_POWER_STATE	Defines an override of the power state of the Ethernet MAC driver.

Description

ETHMAC Driver Configuration Definitions for the template version

These definitions statically define the driver's mode of operation.

File Name

drv_ethmac_config.h

Company

Microchip Technology Inc.

Ethernet PHY Driver Library

This topic describes the Ethernet PHY Driver Library.

Introduction

This library provides a low-level abstraction of the Ethernet PHY Driver Library that is available on the Microchip family of microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, there by hiding differences from one microcontroller variant to another.

Description

This library provides a software abstraction for configuring external Ethernet PHY devices for use with the on-chip PIC32 Ethernet Controller.

Using the Library

The user of this driver is the MPLAB Harmony TCP/IP Stack through its Ethernet MAC driver. This Ethernet PHY driver is not intended as a system wide driver that the application or other system modules may use. It is intended for the sole use of the MPLAB Harmony TCP/IP stack and implements the PHY driver required by the Ethernet MAC.

This topic describes the basic architecture and functionality of the Ethernet PHY driver and is meant for advanced users or TCP/IP Stack driver developers.

Interface Header File: `drv_ethphy.h`

The interface to the Ethernet PHY library is defined in the `drv_ethphy.h` header file, which is included by the MPLAB Harmony TCP/IP stack.

Please refer to the Understanding MPLAB Harmony section for how the library interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the Ethernet PHY Driver Library on Microchip's microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

To understand how this library works you must first understand how an external Ethernet PHY interfaces with the Ethernet Controller. As shown in Figure 1, the PHY has two interfaces, one for managing the PHY, known as the Serial Management Interface (SMI), for configuring the device and a second, known as the Reduced Media Independent Interface (RMII), for transmit and receive data.

Figure 1: Typical External PHY Interface

The block diagram also shows an interrupt signal (nINT) going to a external interrupt pin on the host device and signals going to on-board LEDs to show link state and link activity.

The SMI interface is also known as the MII Management (MIIM) interface. This control interface is standardized for all

PHYs by Clause 22 of the 802.3 standard. It provides up to 32 16-bit registers on the PHY. The following table provides a summary of all 32 registers. Consult the data sheet for the PHY device for the specific bit fields in each register.

Register Address	Register Name	Register Type
0	Control	Basic
1	Status	Basic
2, 3	PHY Identifier	Extended
4	Auto-Negotiation Advertisement	Extended
5	Auto-Negotiation Link Partner Base Page Ability	Extended
6	Auto-Negotiation Expansion	Extended
7	Auto-Negotiation Next Page Transmit	Extended
8	Auto-Negotiation Link Partner Received Next Page	Extended
9	MASTER-SLAVE Control Register	Extended
10	MASTER-SLAVE Status Register	Extended
11-14	Reserved	Extended
15	Extended Status	Reserved
16-31	Vendor Specific	Extended

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Ethernet PHY Driver Library

Library Interface Section	Description
System Level Functions	Routines that integrate the driver into the MPLAB Harmony framework.
Client Level Functions	Open, Close, Link Status, Auto Negotiation.
SMI/MIIM Functions	SMI/MIIM Management Interface.
External PHY Support Functions	Provides the API for PHY support routines that the driver will call when setting up the PHY. The driver library provides support for four PHYs.
Other Functions	Functions that provide software version information.
Data Types and Constants	C language typedefs and enums used by this library.

Configuring the Library

Macros

	Name	Description
	DRV_ETHPHY_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ETHPHY_INDEX	Ethernet PHY static index selection.
	DRV_ETHPHY_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_ETHPHY_PERIPHERAL_ID	Defines an override of the peripheral ID.
	DRV_ETHPHY_NEG_DONE_TMO	Value of the PHY negotiation complete time out as per IEEE 802.3 spec.
	DRV_ETHPHY_NEG_INIT_TMO	Value of the PHY negotiation initiation time out as per IEEE 802.3 spec.
	DRV_ETHPHY_RESET_CLR_TMO	Value of the PHY Reset self clear time out as per IEEE 802.3 spec.

Description

The configuration of the Ethernet PHY Driver Library is based on the file `system_config.h`.

This header file contains the configuration selection for the Ethernet PHY Driver Library. Based on the selections made, the Ethernet PHY Driver Library may support the selected features.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

DRV_ETHPHY_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_CLIENTS_NUMBER 1
```

Description

Ethernet PHY Maximum Number of Clients This definition select the maximum number of clients that the Ethernet PHY driver can support at run time. Not defining it means using a single client.

Remarks

None.

DRV_ETHPHY_INDEX Macro

Ethernet PHY static index selection.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_INDEX DRV_ETHPHY_INDEX_1
```

Description

Ethernet PHY Static Index Selection

This definition selects the Ethernet PHY static index for the driver object reference.

Remarks

This index is required to make a reference to the driver object.

DRV_ETHPHY_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_INSTANCES_NUMBER 1
```

Description

Ethernet PHY hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver. Not defining it means using a static driver.

Remarks

None.

DRV_ETHPHY_PERIPHERAL_ID Macro

Defines an override of the peripheral ID.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_PERIPHERAL_ID ETHPHY_ID_1
```

Description

Ethernet PHY Peripheral ID Selection

Defines an override of the peripheral ID, using macros.

Remarks

Some devices also support ETHPHY_ID_0

DRV_ETHPHY_NEG_DONE_TMO Macro

Value of the PHY negotiation complete time out as per IEEE 802.3 spec.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_NEG_DONE_TMO (2000)
```

Description

Ethernet PHY Negotiation Complete time out

This definition sets the time out of the PHY negotiation complete, in ms.

Remarks

See IEEE 802.3 Clause 28 Table 28-9 autoneg_wait_timer value (max 1s).

DRV_ETHPHY_NEG_INIT_TMO Macro

Value of the PHY negotiation initiation time out as per IEEE 802.3 spec.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_NEG_INIT_TMO (1)
```

Description

Ethernet PHY Negotiation Initiation time out

This definition sets the time out of the PHY negotiation initiation, in ms.

Remarks

None.

DRV_ETHPHY_RESET_CLR_TMO Macro

Value of the PHY Reset self clear time out as per IEEE 802.3 spec.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_RESET_CLR_TMO (500)
```

Description

Ethernet PHY Reset self clear time out

This definition sets the time out of the PHY Reset self clear, in ms.

Remarks

See IEEE 802.3 Clause 22 Table 22-7 and paragraph "22.2.4.1.1 Reset" (max 0.5s)

Building the Library

This section lists the files that are available in the Ethernet PHY Driver Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/driver/ethphy.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
/drv_ethphy.h	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_extphy.c	Basic PHY driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
/src/dynamic/drv_extphy_smsc8700.c	SMSC 8700 PHY implementation file.
/src/dynamic/drv_extphy_smsc8720.c	SMSC 8720 PHY implementation file.
/src/dynamic/drv_extphy_smsc8720.c	SMSC 8740 PHY implementation file.
/src/dynamic/drv_extphy_ip101gr.c	IP101GR PHY implementation file.
/src/dynamic/drv_extphy_dp83640.c	National DP83640 PHY implementation file.
/src/dynamic/drv_extphy_dp83848.c	National DP83848 PHY implementation file.

Module Dependencies

The Ethernet MAC Driver Library depends on the following modules:

- Ethernet MAC Driver Library
- Clock System Service Library
- Ports System Service Library
- Timer System Service Library
- Ethernet Peripheral Library

Library Interface

a) System Level Functions

	Name	Description
	DRV_ETHPHY_Initialize	Initializes the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_Deinitialize	Deinitializes the specified instance of the Ethernet PHY driver module. **Implementation:** Dynamic
	DRV_ETHPHY_NegotiationResultGet	Returns the result of a completed negotiation. **Implementation:** Dynamic
	DRV_ETHPHY_PhyAddressGet	Returns the PHY address. **Implementation:** Dynamic
	DRV_ETHPHY_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_ETHPHY_Status	Provides the current status of the Ethernet PHY driver module. **Implementation:** Dynamic
	DRV_ETHPHY_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic
	DRV_ETHPHY_HWConfigFlagsGet	Returns the current Ethernet PHY hardware MII/RMII and ALTERNATE/DEFAULT configuration flags. **Implementation:** Dynamic
	DRV_ETHPHY_Setup	Initializes Ethernet PHY configuration and set up procedure. **Implementation:** Dynamic

b) Client Level Functions

	Name	Description
	DRV_ETHPHY_ClientStatus	Gets the current client-specific status the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_Close	Closes an opened instance of the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_NegotiationIsComplete	Returns the results of a previously initiated Ethernet PHY negotiation. **Implementation:** Dynamic
	DRV_ETHPHY_Open	Opens the specified Ethernet PHY driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_ETHPHY_Reset	Immediately resets the Ethernet PHY. **Implementation:** Dynamic
	DRV_ETHPHY_RestartNegotiation	Restarts auto-negotiation of the Ethernet PHY link. **Implementation:** Dynamic
	DRV_ETHPHY_ClientOperationAbort	Aborts a current client operation initiated by the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_ClientOperationResult	Gets the result of a client operation initiated by the Ethernet PHY driver. **Implementation:** Dynamic

c) SMI/MIIM Functions

	Name	Description
	DRV_ETHPHY_SMIScanStatusGet	Gets the status of the SMI/MIIM scan data. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanStop	Stops the scan of a previously requested SMI/MIIM register. **Implementation:** Dynamic
	DRV_ETHPHY_SMIClockSet	Sets the SMI/MIIM interface clock. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanStart	Starts the scan of a requested SMI/MIIM register. **Implementation:** Dynamic
	DRV_ETHPHY_SMIRead	Initiates a SMI/MIIM read transaction. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanDataGet	Gets the latest SMI/MIIM scan data result. **Implementation:** Dynamic
	DRV_ETHPHY_SMIStatus	Returns the current status of the SMI/MIIM interface. **Implementation:** Dynamic
	DRV_ETHPHY_SMIWrite	Initiates a SMI/MIIM write transaction. **Implementation:** Dynamic

d) Vendor Functions

	Name	Description
	DRV_ETHPHY_VendorDataGet	Returns the current value of the vendor data. **Implementation:** Dynamic
	DRV_ETHPHY_VendorDataSet	Returns the current value of the vendor data. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIReadResultGet	Reads the result of a previous vendor initiated SMI read transfer with DRV_ETHPHY_VendorSMIReadStart. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIReadStart	Starts a vendor SMI read transfer. Data will be available with DRV_ETHPHY_VendorSMIReadResultGet. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIWriteStart	Starts a vendor SMI write transfer. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	DRV_ETHPHY_CLIENT_STATUS	Identifies the client-specific status of the Ethernet PHY driver.
	DRV_ETHPHY_INIT	Contains all the data necessary to initialize the Ethernet PHY device.
	DRV_ETHPHY_NEGOTIATION_RESULT	Contains all the data necessary to get the Ethernet PHY negotiation result
	DRV_ETHPHY_SETUP	Contains all the data necessary to set up the Ethernet PHY device.
	DRV_ETHPHY_VENDOR_MDIX_CONFIGURE	Pointer to function that configures the MDIX mode for the Ethernet PHY.
	DRV_ETHPHY_VENDOR_MII_CONFIGURE	Pointer to function to configure the Ethernet PHY in one of the MII/RMII operation modes.
	DRV_ETHPHY_VENDOR_SMI_CLOCK_GET	Pointer to a function to return the SMI/MIIM maximum clock speed in Hz of the Ethernet PHY.

	DRV_ETHPHY_INDEX_0	Ethernet PHY driver index definitions.
	DRV_ETHPHY_INDEX_1	This is macro DRV_ETHPHY_INDEX_1.
	DRV_ETHPHY_INDEX_COUNT	Number of valid Ethernet PHY driver indices.
	DRV_ETHPHY_LinkStatusGet	Returns the current link status. **Implementation:** Dynamic
	DRV_ETHPHY_LINK_STATUS	Defines the possible status flags of PHY Ethernet link.
	DRV_ETHPHY_CONFIG_FLAGS	Defines the possible results of Ethernet operations that can succeed or fail
	DRV_ETHPHY_OBJECT	Identifies the interface of a Ethernet PHY driver.
	DRV_ETHPHY_VENDOR_WOL_CONFIGURE	Pointer to a function to configure the PHY WOL functionality

Description

This section describes the Application Programming Interface (API) functions of the Ethernet PHY Driver Library.

Refer to each section for a detailed description.

a) System Level Functions

DRV_ETHPHY_Initialize Function

Initializes the Ethernet PHY driver.

Implementation: Dynamic

File

drv_ethphy.h

C

```
SYS_MODULE_OBJ DRV_ETHPHY_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT *
const init);
```

Returns

- a valid handle to a driver object, if successful.
- SYS_MODULE_OBJ_INVALID if initialization failed.

Description

This function initializes the Ethernet PHY driver, making it ready for clients to open and use it.

Remarks

- This function must be called before any other Ethernet PHY routine is called.
- This function should only be called once during system initialization unless DRV_ETHPHY_Deinitialize is called to deinitialize the driver instance.
- The returned object must be passed as argument to DRV_ETHPHY_Reinitialize, DRV_ETHPHY_Deinitialize, DRV_ETHPHY_Tasks and DRV_ETHPHY_Status routines.

Preconditions

None.

Example

```
DRV_ETHPHY_INIT    init;
SYS_MODULE_OBJ   objectHandle;

// Populate the Ethernet PHY initialization structure
init.phyId  = ETHPHY_ID_0;

// Populate the Ethernet PHY initialization structure
init.phyId  = ETHPHY_ID_2;
init.pPhyObject  = &DRV_ETHPHY_OBJECT_SMSC_LAN8720;

// Do something

objectHandle = DRV_ETHPHY_Initialize(DRV_ETHPHY_INDEX_0, (SYS_MODULE_INIT*)&init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Function

SYS_MODULE_OBJ DRV_ETHPHY_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

DRV_ETHPHY_Deinitialize Function

Deinitializes the specified instance of the Ethernet PHY driver module.

Implementation: Dynamic

File

drv_ethphy.h

C

```
void DRV_ETHPHY_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the specified instance of the Ethernet PHY driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

- Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

Preconditions

The DRV_ETHPHY_Initialize function must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_ETHPHY_Initialize
SYS_STATUS          status;

DRV_ETHPHY_Deinitialize(object);

status = DRV_ETHPHY_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Function

void DRV_ETHPHY_Deinitialize (SYS_MODULE_OBJ object)

DRV_ETHPHY_NegotiationResultGet Function

Returns the result of a completed negotiation.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_NegotiationResultGet(DRV_HANDLE handle,
DRV_ETHPHY_NEGOTIATION_RESULT* pNegResult);
```

Returns

- DRV_ETHPHY_RES_PENDING operation is ongoing
- an DRV_ETHPHY_RESULT error code if the procedure failed.

Description

This function returns the PHY negotiation data gathered after a completed negotiation.

Remarks

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

When operation is completed but negotiation has failed, DRV_ETHPHY_ClientOperationResult will return:

- DRV_ETHPHY_RES_NEGOTIATION_INACTIVE if no negotiation in progress
- DRV_ETHPHY_RES_NEGOTIATION_NOT_STARTED if negotiation not yet started yet (means time out if waitComplete was requested)
- DRV_ETHPHY_RES_NEGOTIATION_ACTIVE if negotiation ongoing

The returned value for the negotiation flags is valid only if the negotiation was completed successfully.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY
- DRV_ETHPHY_RestartNegotiation, and DRV_ETHPHY_NegotiationIsComplete should have been called.

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_NegotiationResultGet(DRV_HANDLE handle, DRV_ETHPHY_NEGOTIATION_RESULT* pNegResult)

DRV_ETHPHY_PhyAddressGet Function

Returns the PHY address.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_PhyAddressGet(DRV_HANDLE handle, int* pPhyAddress);
```

Returns

DRV_ETHPHY_RES_OK - operation successful and the PHY address stored at

DRV_ETHPHY_RES_HANDLE_ERR - passed in handle was invalid pPhyAddress

Description

This function returns the current PHY address as set by the DRV_ETHPHY_Setup procedure.

Remarks

None.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_PhyAddressGet(DRV_HANDLE handle, int* pPhyAddress);

DRV_ETHPHY_Reinitialize Function

Reinitializes the driver and refreshes any associated hardware settings.

Implementation: Dynamic

File

drv_ethphy.h

C

```
void DRV_ETHPHY_Reinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

None.

Description

This function reinitializes the driver and refreshes any associated hardware settings using the initialization data given, but it will not interrupt any ongoing operations.

Remarks

- This function can be called multiple times to reinitialize the module.
- This operation can be used to refresh any supported hardware registers as specified by the initialization data or to change the power state of the module.

Preconditions

The DRV_ETHPHY_Initialize function must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
DRV_ETHPHY_INIT    init;
SYS_MODULE_OBJ   objectHandle;

// Populate the Ethernet PHY initialization structure
init.phyId  = ETHPHY_ID_2;
init.pPhyObject  = &DRV_ETHPHY_OBJECT_SMSC_LAN8720;

DRV_ETHPHY_Reinitialize(objectHandle, (SYS_MODULE_INIT*)&init);

phyStatus = DRV_ETHPHY_Status(objectHandle);
if (SYS_STATUS_BUSY == phyStatus)
{
    // Check again later to ensure the driver is ready
}
else if (SYS_STATUS_ERROR >= phyStatus)
{
    // Handle error
}
```

Function

void DRV_ETHPHY_Reinitialize(SYS_MODULE_OBJ object,

const SYS_MODULE_INIT * const init)

DRV_ETHPHY_Status Function

Provides the current status of the Ethernet PHY driver module.

Implementation: Dynamic

File

drv_ethphy.h

C

```
SYS_STATUS DRV_ETHPHY_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed
- SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed
- SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This function provides the current status of the Ethernet PHY driver module.

Remarks

- Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.
- SYS_STATUS_BUSY - Indicates that the driver is busy with a previous system level operation and cannot start another
- SYS_STATUS_ERROR - Indicates that the driver is in an error state
- Any value less than SYS_STATUS_ERROR is also an error state.
- SYS_MODULE_DEINITIALIZED - Indicates that the driver has been deinitialized
- The this operation can be used to determine when any of the driver's module level operations has completed.
- If the status operation returns SYS_STATUS_BUSY, the a previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.
- The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.
- This function will NEVER block waiting for hardware.
- If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_ETHPHY_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_ETHPHY_Initialize
SYS_STATUS          status;

status = DRV_ETHPHY_Status(object);
if (SYS_STATUS_ERROR >= status)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_ETHPHY_Initialize

Function

SYS_STATUS DRV_ETHPHY_Status (SYS_MODULE_OBJ object)

DRV_ETHPHY_Tasks Function

Maintains the driver's state machine and implements its ISR.

Implementation: Dynamic

File

drv_ethphy.h

C

```
void DRV_ETHPHY_Tasks(SYS_MODULE_OBJ object);
```

Returns

None

Description

This function is used to maintain the driver's internal state machine and implement its ISR for interrupt-driven implementations.

Remarks

- This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks)
- This function will never block or access any resources that may cause it to block.

Preconditions

The DRV_ETHPHY_Initialize routine must have been called for the specified Ethernet PHY driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_ETHPHY_Initialize

while (true)
{
    DRV_ETHPHY_Tasks (object);

    // Do other tasks
}
```

Function

void DRV_ETHPHY_Tasks(SYS_MODULE_OBJ object)

DRV_ETHPHY_HWConfigFlagsGet Function

Returns the current Ethernet PHY hardware MII/RMII and ALTERNATE/DEFAULT configuration flags.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_HWConfigFlagsGet(DRV_HANDLE handle, DRV_ETHPHY_CONFIG_FLAGS*
pFlags);
```

Returns

DRV_ETHPHY_RES_OK - if the configuration flags successfully stored at pFlags DRV_ETHPHY_RESULT error code otherwise

Description

This function returns the current Ethernet PHY hardware MII/RMII and ALTERNATE/DEFAULT configuration flags from the Device Configuration Fuse bits.

Remarks

None.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_ETHPHY_Open)
pFlags	address to store the hardware configuration

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_HWConfigFlagsGet(DRV_HANDLE handle,
DRV_ETHPHY_CONFIG_FLAGS* pFlags)

DRV_ETHPHY_Setup Function

Initializes Ethernet PHY configuration and set up procedure.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_Setup(DRV_HANDLE handle, DRV_ETHPHY_SETUP* pSetUp,
TCPIP_ETH_OPEN_FLAGS* pSetupFlags);
```

Returns

- DRV_ETHPHY_RES_PENDING operation has been scheduled successfully
- an DRV_ETHPHY_RESULT error code if the set up procedure failed.

Description

This function initializes the Ethernet PHY communication. It tries to detect the external Ethernet PHY, to read the capabilities and find a match with the requested features. Then, it programs the Ethernet PHY accordingly.

Remarks

PHY configuration may be a lengthy operation due to active negotiation that the PHY has to perform with the link party. The DRV_ETHPHY_ClientStatus will repeatedly return DRV_ETHPHY_CLIENT_STATUS_BUSY until the set up procedure is complete (unless an error detected at which an error code will be returned immediately).

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_Setup(DRV_HANDLE handle, DRV_ETHPHY_SETUP* pSetUp, TCPIP_ETH_OPEN_FLAGS* pSetupFlags)

b) Client Level Functions

DRV_ETHPHY_ClientStatus Function

Gets the current client-specific status the Ethernet PHY driver.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_CLIENT_STATUS DRV_ETHPHY_ClientStatus(DRV_HANDLE handle);
```

Returns

- DRV_ETHPHY_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the Ethernet PHY driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

This function has to be used to check that a driver operation has completed. It will return DRV_ETHPHY_CLIENT_STATUS_BUSY when an operation is in progress. It will return DRV_ETHPHY_CLIENT_STATUS_READY when the operation has completed.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE phyHandle;  // Returned from DRV_ETHPHY_Open
DRV_ETHPHY_CLIENT_STATUS phyClientStatus;

phyClientStatus = DRV_ETHPHY_ClientStatus(phyHandle);
if(DRV_ETHPHY_CLIENT_STATUS_ERROR >= phyClientStatus)
{
    // Handle the error
}
```

Function

DRV_ETHPHY_CLIENT_STATUS DRV_ETHPHY_ClientStatus(DRV_HANDLE handle)

DRV_ETHPHY_Close Function

Closes an opened instance of the Ethernet PHY driver.

Implementation: Dynamic

File

drv_ethphy.h

C

```
void DRV_ETHPHY_Close(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the Ethernet PHY driver, invalidating the handle.

Remarks

- After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_ETHPHY_Open before the caller may use the driver again.
- Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_ETHPHY_Initialize routine must have been called for the specified Ethernet PHY driver instance.

DRV_ETHPHY_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_ETHPHY_Open

DRV_ETHPHY_Close(handle);
```

Function

void DRV_ETHPHY_Close(DRV_HANDLE handle)

DRV_ETHPHY_NegotiationIsComplete Function

Returns the results of a previously initiated Ethernet PHY negotiation.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_NegotiationIsComplete(DRV_HANDLE handle, bool waitComplete);
```

Returns

- DRV_ETHPHY_RES_PENDING operation is ongoing
- an DRV_ETHPHY_RESULT error code if the procedure failed.

Description

This function returns the results of a previously initiated Ethernet PHY negotiation.

Remarks

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

When operation is completed but negotiation has failed, DRV_ETHPHY_ClientOperationResult will return:

- DRV_ETHPHY_RES_NEGOTIATION_INACTIVE if no negotiation in progress
- DRV_ETHPHY_RES_NEGOTIATION_NOT_STARTED if negotiation not yet started yet (means time out if waitComplete was requested)
- DRV_ETHPHY_RES_NEGOTIATION_ACTIVE if negotiation ongoing

(means time out if waitComplete was requested).

See also DRV_ETHPHY_NegotiationResultGet.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY
- DRV_ETHPHY_RestartNegotiation should have been called.

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_NegotiationIsComplete(DRV_HANDLE handle, bool waitComplete)

DRV_ETHPHY_Open Function

Opens the specified Ethernet PHY driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_HANDLE DRV_ETHPHY_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

- valid open-instance handle if successful (a number identifying both the caller and the module instance).
- DRV_HANDLE_INVALID if an error occurs

Description

This function opens the specified Ethernet PHY driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_ETHPHY_Close routine is called.

This function will NEVER block waiting for hardware.

The intent parameter is not used. The PHY driver implements a non-blocking behavior.

Preconditions

The DRV_ETHPHY_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_ETHPHY_Open(DRV_ETHPHY_INDEX_0, 0);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Function

DRV_HANDLE DRV_ETHPHY_Open(const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT intent)

DRV_ETHPHY_Reset Function

Immediately resets the Ethernet PHY.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_Reset(DRV_HANDLE handle, bool waitComplete);
```

Returns

- DRV_ETHPHY_RES_PENDING for ongoing, in progress operation
- DRV_ETHPHY_RES_OPERATION_ERR - invalid parameter or operation in the current context

Description

This function immediately resets the Ethernet PHY, optionally waiting for a reset to complete.

Remarks

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

When operation is completed but failed, DRV_ETHPHY_ClientOperationResult will return:

- DRV_ETHPHY_RES_DTCT_ERR if the PHY failed to respond

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_Reset(DRV_HANDLE handle, bool waitComplete)

© 2013-2015 Microchip Technology Inc.

DRV_ETHPHY_RestartNegotiation Function

Restarts auto-negotiation of the Ethernet PHY link.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_RestartNegotiation(DRV_HANDLE handle);
```

Returns

- DRV_ETHPHY_RES_PENDING operation has been scheduled successfully
- an DRV_ETHPHY_RESULT error code if the procedure failed.

Description

This function restarts auto-negotiation of the Ethernet PHY link.

Remarks

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_RestartNegotiation(DRV_HANDLE handle)

DRV_ETHPHY_ClientOperationAbort Function

Aborts a current client operation initiated by the Ethernet PHY driver.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_ClientOperationAbort(DRV_HANDLE handle);
```

Returns

- DRV_ETHPHY_RESULT value describing the current operation result: DRV_ETHPHY_RES_OK for success; operation has been aborted an DRV_ETHPHY_RESULT error code if the operation failed.

Description

Aborts a current client operation initiated by the Ethernet PHY driver.

Remarks

None

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid opened device handle.
- A driver operation was started

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_ClientOperationAbort(DRV_HANDLE handle)

DRV_ETHPHY_ClientOperationResult Function

Gets the result of a client operation initiated by the Ethernet PHY driver.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_ClientOperationResult(DRV_HANDLE handle);
```

Returns

- DRV_ETHPHY_RESULT value describing the current operation result: DRV_ETHPHY_RES_OK for success; operation has been completed successfully DRV_ETHPHY_RES_PENDING operation is in progress an DRV_ETHPHY_RESULT error code if the operation failed.

Description

Returns the result of a client operation initiated by the Ethernet PHY driver.

Remarks

This function will not block for hardware access and will immediately return the current status.

This function returns the result of the last driver operation. It will return DRV_ETHPHY_RES_PENDING if an operation is still in progress. Otherwise a DRV_ETHPHY_RESULT describing the operation outcome.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid opened device handle.
- A driver operation was started and completed

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_ClientOperationResult(DRV_HANDLE handle)

c) SMI/MIIM Functions

DRV_ETHPHY_SMIScanStatusGet Function

Gets the status of the SMI/MIIM scan data.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStatusGet(DRV_HANDLE handle);
```

Returns

DRV_ETHPHY_RES_OPERATION_ERR - no scan operation currently in progress

DRV_ETHPHY_RES_OK - scan data is available

DRV_ETHPHY_RES_PENDING - scan data is not yet available

< 0 - an error has occurred and the operation could not be completed

Description

This function gets the status of the SMI/MIIM scan data.

Remarks

None.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY
- DRV_ETHPHY_SMIScanStart() has been called.

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStatusGet(DRV_HANDLE handle)

DRV_ETHPHY_SMIScanStop Function

Stops the scan of a previously requested SMI/MIIM register.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStop(DRV_HANDLE handle);
```

Returns

DRV_ETHPHY_RES_OPERATION_ERR - no scan operation currently in progress

DRV_ETHPHY_RES_OK - the scan transaction has been stopped successfully < 0 - an error has occurred and the operation could not be completed

Description

This function stops the current scan of a SMI/MIIM register.

Remarks

None.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY
- DRV_ETHPHY_SMIScanStart was called to start a scan

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStop(DRV_HANDLE handle)

DRV_ETHPHY_SMIClockSet Function

Sets the SMI/MIIM interface clock.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIClockSet(DRV_HANDLE handle, uint32_t hostClock, uint32_t
maxSMIClock);
```

Returns

DRV_ETHPHY_RES_HANDLE_ERR - passed in handle was invalid

DRV_ETHPHY_RES_OK - operation successful

Description

This function sets SMI/MIIM interface clock base on host clock and maximum supported SMI/MIIM interface clock speed.

Remarks

None.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIClockSet(DRV_HANDLE handle,

uint32_t hostClock,

uint32_t maxSMIClock)

DRV_ETHPHY_SMIScanStart Function

Starts the scan of a requested SMI/MIIM register.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStart(DRV_HANDLE handle, unsigned int rIx);
```

Returns

DRV_ETHPHY_RES_PENDING - the scan transaction was initiated and is ongoing < 0 - an error has occurred and the operation could not be completed

Description

This function starts the scan of a requested SMI/MIIM register.

Remarks

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

However, the client status will always be DRV_ETHPHY_CLIENT_STATUS_BUSY and the client result will always show DRV_ETHPHY_RES_PENDING for as long as the scan is active. Use DRV_ETHPHY_SMIScanStop() to stop a scan in progress. Use DRV_ETHPHY_SMIScanStatusGet() to check is there is scan data available. Use DRV_ETHPHY_SMIScanDataGet() to retrieve the scan data.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStart(DRV_HANDLE handle,

unsigned int rIx)

DRV_ETHPHY_SMIRead Function

Initiates a SMI/MIIM read transaction.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIRead(DRV_HANDLE handle, unsigned int rIx, uint16_t* pSmiRes,
int phyAdd);
```

Returns

DRV_ETHPHY_RES_PENDING - the transaction was initiated and is ongoing < 0 - an error has occurred and the operation could not be completed

Description

This function initiates a SMI/MIIM read transaction for a given PHY register.

Remarks

In most situations the PHY address to be used for this function should be the one returned by DRV_ETHPHY_PhyAddressGet(). However this function allows using a different PHY address for advanced operation.

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid opened device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIRead(DRV_HANDLE handle, unsigned int rIx, uint16_t* pSmiRes, int phyAdd)

DRV_ETHPHY_SMIScanDataGet Function

Gets the latest SMI/MIIM scan data result.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanDataGet(DRV_HANDLE handle, uint16_t* pScanRes);
```

Returns

DRV_ETHPHY_RES_OPERATION_ERR - no scan operation currently in progress

DRV_ETHPHY_RES_OK - scan data is available and stored at pScanRes DRV_ETHPHY_RES_PENDING - scan data is not yet available

< 0 - an error has occurred and the operation could not be completed

Description

This function gets the latest SMI/MIIM scan data result.

Remarks

None.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY
- DRV_ETHPHY_SMIScanStart() has been called
- Data is available if DRV_ETHPHY_SMIScanStatusGet() previously returned DRV_ETHPHY_RES_OK

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanDataGet(DRV_HANDLE handle, uint16_t* pScanRes)

DRV_ETHPHY_SMIStatus Function

Returns the current status of the SMI/MIIM interface.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIStatus(DRV_HANDLE handle);
```

Returns

- DRV_ETHPHY_RES_BUSY - if the SMI/MIIM interface is busy

- DRV_ETHPHY_RES_OK - if the SMI/MIIM is not busy

< 0 - an error has occurred and the operation could not be completed

Description

This function checks if the SMI/MIIM interface is busy with a transaction.

Remarks

This function is info only and returns the momentary status of the SMI bus. Even if the bus is free there is no guarantee it will be free later on especially if the driver is on going some operation.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIStatus(DRV_HANDLE handle)

DRV_ETHPHY_SMIWrite Function

Initiates a SMI/MIIM write transaction.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIWrite(DRV_HANDLE handle, unsigned int rIx, uint16_t wData, int
phyAdd, bool waitComplete);
```

Returns

DRV_ETHPHY_RES_OK - the write transaction has been scheduled/completed successfully DRV_ETHPHY_RES_PENDING - the transaction was initiated and is ongoing < 0 - an error has occurred and the operation could not be completed

Description

This function initiates a SMI/MIIM write transaction for a given PHY register.

Remarks

In most situations the PHY address to be used for this function should be the one returned by DRV_ETHPHY_PhyAddressGet(). However this function allows using a different PHY address for advanced operation.

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIWrite(DRV_HANDLE handle, unsigned int rIx, uint16_t wData, int phyAdd, bool waitComplete)

d) Vendor Functions

DRV_ETHPHY_VendorDataGet Function

Returns the current value of the vendor data.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_VendorDataGet(DRV_HANDLE handle, uint32_t* pVendorData);
```

Returns

DRV_ETHPHY_RES_OK - if the vendor data is stored at the pVendorData address

DRV_ETHPHY_RES_HANDLE_ERR - handle error

Description

This function returns the current value of the vendor data. Each DRV_ETHPHY client object maintains data that could be used for vendor specific operations. This routine allows retrieving of the vendor specific data.

Remarks

The PHY driver will clear the vendor specific data before any call to a vendor specific routine. Otherwise the PHY driver functions do not touch this value.

The DRV_ETHPHY_VendorDataSet can be used for writing data into this field.

Currently only a 32 bit value is supported.

The function is intended for implementing vendor specific functions, like DRV_EXTPHY_MIIConfigure and DRV_EXTPHY_MDIXConfigure, that need a way of maintaining their own data and state machine.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_VendorDataGet(DRV_HANDLE handle, uint32_t* pVendorData)

DRV_ETHPHY_VendorDataSet Function

Returns the current value of the vendor data.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_VendorDataSet(DRV_HANDLE handle, uint32_t vendorData);
```

Returns

DRV_ETHPHY_RES_OK - if the vendor data is stored in the client object

DRV_ETHPHY_RES_HANDLE_ERR - handle error

Description

This function returns the current value of the vendor data. Each DRV_ETHPHY client object maintains data that could be used for vendor specific operations. This routine allows retrieving of the vendor specific data.

Remarks

The PHY driver will clear the vendor specific data before any call to a vendor specific routine. Otherwise the PHY driver functions do not touch this value.

The DRV_ETHPHY_VendorDataGet can be used for reading data into this field.

Currently only a 32 bit value is supported.

The function is intended for implementing vendor specific functions, like DRV_EXTPHY_MIIConfigure and DRV_EXTPHY_MDIXConfigure, that need a way of maintaining their own data and state machine.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_VendorDataSet(DRV_HANDLE handle, uint32_t vendorData)

DRV_ETHPHY_VendorSMIReadResultGet Function

Reads the result of a previous vendor initiated SMI read transfer with DRV_ETHPHY_VendorSMIReadStart.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIReadResultGet(DRV_HANDLE handle, uint16_t* pSmiRes);
```

Returns

DRV_ETHPHY_RES_OK - transaction complete and result deposited at pSmiRes.

DRV_ETHPHY_RES_PENDING - if the vendor transaction is still ongoing The call needs to be retried.

< 0 - some error and the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure has to return error to be aborted by the DRV_ETHPHY_Setup

Description

This function will return the data of a SMI read transfer.

Remarks

The function is intended for implementing vendor SMI transfers within DRV_EXTPHY_MIIConfigure and DRV_EXTPHY_MDIXConfigure.

It has to be called from within the DRV_EXTPHY_MIIConfigure or DRV_EXTPHY_MDIXConfigure functions (which are called, in turn, by the DRV_ETHPHY_Setup procedure) otherwise the call will fail.

The DRV_ETHPHY_RES_OK and DRV_ETHPHY_RES_PENDING significance is changed from the general driver API.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup is in progress and configures the PHY
- The vendor implementation of the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure is running and a SMI transfer is needed
- DRV_ETHPHY_VendorSMIReadStart should have been called to initiate a transfer

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIReadResultGet(DRV_HANDLE handle, uint16_t* pSmiRes)

DRV_ETHPHY_VendorSMIReadStart Function

Starts a vendor SMI read transfer. Data will be available with DRV_ETHPHY_VendorSMIReadResultGet.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIReadStart(DRV_HANDLE handle, uint16_t rIx, int
phyAddress);
```

Returns

DRV_ETHPHY_RES_OK - the vendor transaction is started DRV_ETHPHY_VendorSMIReadResultGet() needs to be called for the transaction to complete and to retrieve the result

DRV_ETHPHY_RES_PENDING - the SMI bus is busy and the call needs to be retried

< 0 - some error and the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure has to return error to be aborted by the DRV_ETHPHY_Setup

Description

This function will start a SMI read transfer.

Remarks

The function is intended for implementing vendor SMI transfers within DRV_EXTPHY_MIIConfigure and DRV_EXTPHY_MDIXConfigure.

It has to be called from within the DRV_EXTPHY_MIIConfigure or DRV_EXTPHY_MDIXConfigure functions (which are called, in turn, by the DRV_ETHPHY_Setup procedure) otherwise the call will fail.

The DRV_ETHPHY_RES_OK and DRV_ETHPHY_RES_PENDING significance is changed from the general driver API.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup is in progress and configures the PHY
- The vendor implementation of the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure is running and a SMI transfer is needed

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIReadStart(DRV_HANDLE handle, uint16_t rIx, int phyAddress)

DRV_ETHPHY_VendorSMIWriteStart Function

Starts a vendor SMI write transfer.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIWriteStart(DRV_HANDLE handle, uint16_t rIx, uint16_t
wData, int phyAddress);
```

Returns

DRV_ETHPHY_RES_OK - if the vendor SMI write transfer is started

DRV_ETHPHY_RES_PENDING - the SMI bus was busy and the call needs to be retried

< 0 - some error and the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure has to return error to be aborted by the DRV_ETHPHY_Setup

Description

This function will start a SMI write transfer.

Remarks

The function is intended for implementing vendor SMI transfers within DRV_EXTPHY_MIIConfigure and DRV_EXTPHY_MDIXConfigure.

It has to be called from within the DRV_EXTPHY_MIIConfigure or DRV_EXTPHY_MDIXConfigure functions (which are called, in turn, by the DRV_ETHPHY_Setup procedure) otherwise the call will fail.

The DRV_ETHPHY_RES_OK and DRV_ETHPHY_RES_PENDING significance is changed from the general driver API.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup is in progress and configures the PHY
- The vendor implementation of the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure is running and a SMI transfer is needed

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIWriteStart(DRV_HANDLE handle, uint16_t rIx, uint16_t wData, int phyAddress)

e) Other Functions

f) Data Types and Constants

DRV_ETHPHY_CLIENT_STATUS Enumeration

Identifies the client-specific status of the Ethernet PHY driver.

File

drv_ethphy.h

C

```
typedef enum {
  DRV_ETHPHY_CLIENT_STATUS_ERROR,
  DRV_ETHPHY_CLIENT_STATUS_CLOSED,
  DRV_ETHPHY_CLIENT_STATUS_BUSY,
  DRV_ETHPHY_CLIENT_STATUS_READY
} DRV_ETHPHY_CLIENT_STATUS;
```

Members

Members	Description
DRV_ETHPHY_CLIENT_STATUS_ERROR	Unspecified error condition
DRV_ETHPHY_CLIENT_STATUS_CLOSED	Client is not open
DRV_ETHPHY_CLIENT_STATUS_BUSY	An operation is currently in progress
DRV_ETHPHY_CLIENT_STATUS_READY	Up and running, no operations running

Description

Ethernet PHY Driver Client Status

This enumeration identifies the client-specific status of the Ethernet PHY driver.

Remarks

None.

DRV_ETHPHY_INIT Structure

Contains all the data necessary to initialize the Ethernet PHY device.

File

drv_ethphy.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  ETH_MODULE_ID ethphyId;
  const DRV_ETHPHY_OBJECT* pPhyObject;
} DRV_ETHPHY_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
ETH_MODULE_ID ethphyId;	Identifies peripheral (PLIB-level) ID
const DRV_ETHPHY_OBJECT* pPhyObject;	Non-volatile pointer to the PHY object providing vendor functions for this PHY

Description

Ethernet PHY Device Driver Initialization Data

This data structure contains all the data necessary to initialize the Ethernet PHY device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_ETHPHY_Initialize routine.

DRV_ETHPHY_NEGOTIATION_RESULT Structure

Contains all the data necessary to get the Ethernet PHY negotiation result

File

drv_ethphy.h

C

```
typedef struct {
  DRV_ETHPHY_LINK_STATUS linkStatus;
  TCPIP_ETH_OPEN_FLAGS linkFlags;
  TCPIP_ETH_PAUSE_TYPE pauseType;
} DRV_ETHPHY_NEGOTIATION_RESULT;
```

Members

Members	Description
DRV_ETHPHY_LINK_STATUS linkStatus;	link status after a completed negotiation
TCPIP_ETH_OPEN_FLAGS linkFlags;	the negotiation result flags
TCPIP_ETH_PAUSE_TYPE pauseType;	pause type supported by the link partner

Description

Ethernet PHY Device Driver Negotiation result Data

Contains all the data necessary to get the Ethernet PHY negotiation result

Remarks

A pointer to a structure of this format must be passed into the DRV_ETHPHY_NegotiationResultGet routine.

DRV_ETHPHY_SETUP Structure

Contains all the data necessary to set up the Ethernet PHY device.

File

drv_ethphy.h

C

```
typedef struct {
  int phyAddress;
  TCPIP_ETH_OPEN_FLAGS openFlags;
  DRV_ETHPHY_CONFIG_FLAGS configFlags;
  TCPIP_ETH_PAUSE_TYPE macPauseType;
} DRV_ETHPHY_SETUP;
```

Members

Members	Description
int phyAddress;	the address the PHY is configured for
TCPIP_ETH_OPEN_FLAGS openFlags;	the capability flags: FD/HD, 100/100Mbps, etc.
DRV_ETHPHY_CONFIG_FLAGS configFlags;	configuration flags: MII/RMII, I/O setup
TCPIP_ETH_PAUSE_TYPE macPauseType;	MAC requested pause type

Description

Ethernet PHY Device Driver Set up Data

This data structure contains all the data necessary to configure the Ethernet PHY device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_ETHPHY_Setup routine.

DRV_ETHPHY_VENDOR_MDIX_CONFIGURE Type

Pointer to function that configures the MDIX mode for the Ethernet PHY.

File

drv_ethphy.h

C

```
typedef DRV_ETHPHY_RESULT (* DRV_ETHPHY_VENDOR_MDIX_CONFIGURE)(DRV_HANDLE handle,
TCPIP_ETH_OPEN_FLAGS oFlags);
```

Returns

- DRV_ETHPHY_RES_OK - if success, operation complete
- DRV_ETHPHY_RES_PENDING - if function needs to be called again

< 0 - on failure: configuration not supported or some other error

Description

Pointer To Function: typedef DRV_ETHPHY_RESULT (* DRV_ETHPHY_VENDOR_MDIX_CONFIGURE) (DRV_HANDLE handle, TCPIP_ETH_OPEN_FLAGS oFlags);

This type describes a pointer to a function that configures the MDIX mode for the Ethernet PHY. This configuration function is PHY specific and every PHY driver has to provide their own implementation.

Remarks

The PHY driver consists of 2 modules:

- the main PHY driver which uses standard IEEE PHY registers
- the vendor specific functionality

This function provides vendor specific functionality. Every PHY driver has to expose this vendor specific function as part of its interface.

Traditionally the name used for this function is DRV_EXTPHY_MDIXConfigure but any name can be used.

The function can use all the vendor specific functions to store/retrieve specific data or start SMI transactions (see Vendor Interface Routines).

The function should not block but return DRV_ETHPHY_RES_PENDING if waiting for SMI transactions.

Preconditions

Communication to the PHY should have been established.

DRV_ETHPHY_VENDOR_MII_CONFIGURE Type

Pointer to function to configure the Ethernet PHY in one of the MII/RMII operation modes.

File

drv_ethphy.h

C

```
typedef DRV_ETHPHY_RESULT (* DRV_ETHPHY_VENDOR_MII_CONFIGURE)(DRV_HANDLE handle,
DRV_ETHPHY_CONFIG_FLAGS cFlags);
```

Returns

- DRV_ETHPHY_RES_OK - if success, operation complete
- DRV_ETHPHY_RES_PENDING - if function needs to be called again

< 0 - on failure: configuration not supported or some other error

Description

Pointer To Function: typedef DRV_ETHPHY_RESULT (* DRV_ETHPHY_VENDOR_MII_CONFIGURE) (DRV_HANDLE handle, DRV_ETHPHY_CONFIG_FLAGS cFlags);

This type describes a pointer to a function that configures the Ethernet PHY in one of the MII/RMII operation modes. This configuration function is PHY specific and every PHY driver has to provide their own implementation.

Remarks

The PHY driver consists of 2 modules:

- the main PHY driver which uses standard IEEE PHY registers
- the vendor specific functionality

This function provides vendor specific functionality. Every PHY driver has to expose this vendor specific function as part of its interface.

Traditionally the name used for this function is DRV_EXTPHY_MIIConfigure but any name can be used.

The PHY driver will call the vendor set up functions after the communication to the PHY has been established.

The function can use all the vendor specific functions to store/retrieve specific data or start SMI transactions (see Vendor Interface Routines).

The function should not block but return DRV_ETHPHY_RES_PENDING if waiting for SMI transactions.

Preconditions

Communication to the PHY should have been established.

DRV_ETHPHY_VENDOR_SMI_CLOCK_GET Type

Pointer to a function to return the SMI/MIIM maximum clock speed in Hz of the Ethernet PHY.

File

drv_ethphy.h

C

```
typedef unsigned int (* DRV_ETHPHY_VENDOR_SMI_CLOCK_GET)(DRV_HANDLE handle);
```

Returns

The maximum SMI/MIIM clock speed as an unsigned integer.

Description

Pointer to Function: typedef unsigned int (* DRV_ETHPHY_VENDOR_SMI_CLOCK_GET) (DRV_HANDLE handle);

This type describes a pointer to a function that returns the SMI/MIIM maximum clock speed in Hz of the Ethernet PHY. This configuration function is PHY specific and every PHY driver has to provide their own implementation.

Remarks

The PHY driver consists of 2 modules:

- the main PHY driver which uses standard IEEE PHY registers
- the vendor specific functionality

This function provides vendor specific functionality. Every PHY driver has to expose this vendor specific function as part of its interface.

This value is PHY specific. All PHYs are requested to support 2.5 MHz.

Traditionally the name used for this function is DRV_EXTPHY_SMIClockGet but any name can be used.

The PHY driver will call the vendor set up functions after the communication to the PHY has been established.

The function should not block but return immediately. The function cannot start SMI transactions and cannot use the vendor specific functions to store/retrieve specific data (see Vendor Interface Routines).

Preconditions

Communication to the PHY should have been established.

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_ETHPHY_Open)

DRV_ETHPHY_INDEX_0 Macro

Ethernet PHY driver index definitions.

File

drv_ethphy.h

C

```
#define DRV_ETHPHY_INDEX_0 0
```

Description

Ethernet PHY Driver Module Index Numbers

These constants provide the Ethernet PHY driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_ETHPHY_Initialize and DRV_ETHPHY_Open routines to identify the driver instance in use.

DRV_ETHPHY_INDEX_1 Macro

File

drv_ethphy.h

C

```
#define DRV_ETHPHY_INDEX_1 1
```

Description

This is macro DRV_ETHPHY_INDEX_1.

DRV_ETHPHY_INDEX_COUNT Macro

Number of valid Ethernet PHY driver indices.

File

drv_ethphy.h

C

```
#define DRV_ETHPHY_INDEX_COUNT 1
```

Description

Ethernet PHY Driver Module Index Count

This constant identifies the number of valid Ethernet PHY driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from part-specific header files defined as part of the peripheral libraries.

DRV_ETHPHY_LinkStatusGet Function

Returns the current link status.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_LinkStatusGet(DRV_HANDLE handle, DRV_ETHPHY_LINK_STATUS*
pLinkStat, bool refresh);
```

Returns

- DRV_ETHPHY_RES_PENDING for ongoing, in progress operation
- an DRV_ETHPHY_RESULT error code if the link status get procedure failed.

Description

This function returns the current link status.

Remarks

This function reads the Ethernet PHY to get current link status. If refresh is specified then, if the link is down a second read will be performed to return the current link status.

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_LinkStatusGet(DRV_HANDLE handle, DRV_ETHPHY_LINK_STATUS* pLinkStat, bool refresh)

DRV_ETHPHY_LINK_STATUS Enumeration

Defines the possible status flags of PHY Ethernet link.

File

drv_ethphy.h

C

```
typedef enum {
  DRV_ETHPHY_LINK_ST_DOWN,
  DRV_ETHPHY_LINK_ST_UP,
  DRV_ETHPHY_LINK_ST_LP_NEG_UNABLE,
  DRV_ETHPHY_LINK_ST_REMOTE_FAULT,
  DRV_ETHPHY_LINK_ST_PDF,
  DRV_ETHPHY_LINK_ST_LP_PAUSE,
  DRV_ETHPHY_LINK_ST_LP_ASM_DIR,
  DRV_ETHPHY_LINK_ST_NEG_TMO,
  DRV_ETHPHY_LINK_ST_NEG_FATAL_ERR
} DRV_ETHPHY_LINK_STATUS;
```

Members

Members	Description
DRV_ETHPHY_LINK_ST_DOWN	No connection to the LinkPartner
DRV_ETHPHY_LINK_ST_UP	Link is up
DRV_ETHPHY_LINK_ST_LP_NEG_UNABLE	LP non negotiation able
DRV_ETHPHY_LINK_ST_REMOTE_FAULT	LP fault during negotiation
DRV_ETHPHY_LINK_ST_PDF	Parallel Detection Fault encountered (when DRV_ETHPHY_LINK_ST_LP_NEG_UNABLE)
DRV_ETHPHY_LINK_ST_LP_PAUSE	LP supports symmetric pause
DRV_ETHPHY_LINK_ST_LP_ASM_DIR	LP supports asymmetric TX/RX pause operation
DRV_ETHPHY_LINK_ST_NEG_TMO	LP not there
DRV_ETHPHY_LINK_ST_NEG_FATAL_ERR	An unexpected fatal error occurred during the negotiation

Description

Ethernet PHY Device Link Status Codes

This enumeration defines the flags describing the status of the PHY Ethernet link.

Remarks

Multiple flags can be set.

DRV_ETHPHY_CONFIG_FLAGS Enumeration

Defines the possible results of Ethernet operations that can succeed or fail

File

drv_ethphy.h

C

```
typedef enum {
  DRV_ETHPHY_CFG_RMII,
  DRV_ETHPHY_CFG_MII,
  DRV_ETHPHY_CFG_ALTERNATE,
  DRV_ETHPHY_CFG_DEFAULT,
  DRV_ETHPHY_CFG_AUTO
} DRV_ETHPHY_CONFIG_FLAGS;
```

Members

Members	Description
DRV_ETHPHY_CFG_RMII	RMII data interface in configuration fuses.
DRV_ETHPHY_CFG_MII	MII data interface in configuration fuses.
DRV_ETHPHY_CFG_ALTERNATE	Configuration fuses is ALT
DRV_ETHPHY_CFG_DEFAULT	Configuration fuses is DEFAULT
DRV_ETHPHY_CFG_AUTO	Use the fuses configuration to detect if you are RMII/MII and ALT/DEFAULT configuration

Description

Ethernet PHY Driver Operation Result *

PHY Driver Operation Result Codes

This enumeration defines the possible results of any of the PHY driver operations that have the possibility of failing. This result should be checked to ensure that the operation achieved the desired result.

DRV_ETHPHY_OBJECT Structure

Identifies the interface of a Ethernet PHY driver.

File

drv_ethphy.h

C

```
typedef struct {
  DRV_ETHPHY_VENDOR_MII_CONFIGURE miiConfigure;
  DRV_ETHPHY_VENDOR_MDIX_CONFIGURE mdixConfigure;
  DRV_ETHPHY_VENDOR_SMI_CLOCK_GET smiClockGet;
  DRV_ETHPHY_VENDOR_WOL_CONFIGURE wolConfigure;
} DRV_ETHPHY_OBJECT;
```

Members

Members	Description
DRV_ETHPHY_VENDOR_MII_CONFIGURE miiConfigure;	PHY driver function to configure the operation mode: MII/RMII
DRV_ETHPHY_VENDOR_MDIX_CONFIGURE mdixConfigure;	PHY driver function to configure the MDIX mode
DRV_ETHPHY_VENDOR_SMI_CLOCK_GET smiClockGet;	PHY driver function to get the SMI clock rate
DRV_ETHPHY_VENDOR_WOL_CONFIGURE wolConfigure;	PHY driver function to configure the WOL functionality

Description

Ethernet PHY Driver Object

This data structure identifies the required interface of the Ethernet PHY driver. Any PHY driver has to export this interface.

Remarks

None.

DRV_ETHPHY_VENDOR_WOL_CONFIGURE Type

Pointer to a function to configure the PHY WOL functionality

File

drv_ethphy.h

C

```
typedef void (* DRV_ETHPHY_VENDOR_WOL_CONFIGURE)(DRV_HANDLE handle, unsigned char bAddr[]);
```

Returns

None

Description

Pointer to Function: typedef void (* DRV_ETHPHY_VENDOR_WOL_CONFIGURE) (DRV_HANDLE handle, unsigned char bAddr[]);

This type describes a pointer to a function that configures the PHY WOL functionality of the Ethernet PHY. Configures the WOL of the PHY with a Source MAC address or a 6 byte magic packet mac address.

This configuration function is PHY specific and every PHY driver has to provide their own implementation.

Remarks

The PHY driver consists of 2 modules:

- the main PHY driver which uses standard IEEE PHY registers
- the vendor specific functionality

This function provides vendor specific functionality. Every PHY driver has to expose this vendor specific function as part of its interface.

Traditionally the name used for this function is DRV_EXTPHY_WOLConfiguration but any name can be used.

The PHY driver will call the vendor set up functions after the communication to the PHY has been established.

The function can use all the vendor specific functions to store/retrieve specific data or start SMI transactions (see Vendor Interface Routines).

The function should not block but return DRV_ETHPHY_RES_PENDING if waiting for SMI transactions.

This feature is not currently supported for all PHYs.

Preconditions

Communication to the PHY should have been established.

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_ETHPHY_Open)
bAddr[]	Source Mac Address, or a Magic Packet MAC address

Files

Files

Name	Description
drv_ethphy.h	Ethernet ETHPHY Device Driver Interface File
drv_ethphy_config.h	Ethernet PHY driver configuration definitions template.

Description

This section lists the source and header files used by the Ethernet PHY Driver Library.

drv_ethphy.h

Ethernet ETHPHY Device Driver Interface File

Enumerations

	Name	Description
	DRV_ETHPHY_CLIENT_STATUS	Identifies the client-specific status of the Ethernet PHY driver.
	DRV_ETHPHY_CONFIG_FLAGS	Defines the possible results of Ethernet operations that can succeed or fail
	DRV_ETHPHY_LINK_STATUS	Defines the possible status flags of PHY Ethernet link.

Functions

	Name	Description
	DRV_ETHPHY_ClientOperationAbort	Aborts a current client operation initiated by the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_ClientOperationResult	Gets the result of a client operation initiated by the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_ClientStatus	Gets the current client-specific status the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_Close	Closes an opened instance of the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_Deinitialize	Deinitializes the specified instance of the Ethernet PHY driver module. **Implementation:** Dynamic
	DRV_ETHPHY_HWConfigFlagsGet	Returns the current Ethernet PHY hardware MII/RMII and ALTERNATE/DEFAULT configuration flags. **Implementation:** Dynamic
	DRV_ETHPHY_Initialize	Initializes the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_LinkStatusGet	Returns the current link status. **Implementation:** Dynamic
	DRV_ETHPHY_NegotiationIsComplete	Returns the results of a previously initiated Ethernet PHY negotiation. **Implementation:** Dynamic
	DRV_ETHPHY_NegotiationResultGet	Returns the result of a completed negotiation. **Implementation:** Dynamic

	DRV_ETHPHY_Open	Opens the specified Ethernet PHY driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_ETHPHY_PhyAddressGet	Returns the PHY address. **Implementation:** Dynamic
	DRV_ETHPHY_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_ETHPHY_Reset	Immediately resets the Ethernet PHY. **Implementation:** Dynamic
	DRV_ETHPHY_RestartNegotiation	Restarts auto-negotiation of the Ethernet PHY link. **Implementation:** Dynamic
	DRV_ETHPHY_Setup	Initializes Ethernet PHY configuration and set up procedure. **Implementation:** Dynamic
	DRV_ETHPHY_SMIClockSet	Sets the SMI/MIIM interface clock. **Implementation:** Dynamic
	DRV_ETHPHY_SMIRead	Initiates a SMI/MIIM read transaction. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanDataGet	Gets the latest SMI/MIIM scan data result. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanStart	Starts the scan of a requested SMI/MIIM register. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanStatusGet	Gets the status of the SMI/MIIM scan data. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanStop	Stops the scan of a previously requested SMI/MIIM register. **Implementation:** Dynamic
	DRV_ETHPHY_SMIStatus	Returns the current status of the SMI/MIIM interface. **Implementation:** Dynamic
	DRV_ETHPHY_SMIWrite	Initiates a SMI/MIIM write transaction. **Implementation:** Dynamic
	DRV_ETHPHY_Status	Provides the current status of the Ethernet PHY driver module. **Implementation:** Dynamic
	DRV_ETHPHY_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic
	DRV_ETHPHY_VendorDataGet	Returns the current value of the vendor data. **Implementation:** Dynamic
	DRV_ETHPHY_VendorDataSet	Returns the current value of the vendor data. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIReadResultGet	Reads the result of a previous vendor initiated SMI read transfer with DRV_ETHPHY_VendorSMIReadStart. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIReadStart	Starts a vendor SMI read transfer. Data will be available with DRV_ETHPHY_VendorSMIReadResultGet. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIWriteStart	Starts a vendor SMI write transfer. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_ETHPHY_INDEX_0	Ethernet PHY driver index definitions.
	DRV_ETHPHY_INDEX_1	This is macro DRV_ETHPHY_INDEX_1.
	DRV_ETHPHY_INDEX_COUNT	Number of valid Ethernet PHY driver indices.

Structures

	Name	Description
	DRV_ETHPHY_INIT	Contains all the data necessary to initialize the Ethernet PHY device.
	DRV_ETHPHY_NEGOTIATION_RESULT	Contains all the data necessary to get the Ethernet PHY negotiation result
	DRV_ETHPHY_OBJECT	Identifies the interface of a Ethernet PHY driver.
	DRV_ETHPHY_SETUP	Contains all the data necessary to set up the Ethernet PHY device.

Types

	Name	Description
	DRV_ETHPHY_VENDOR_MDIX_CONFIGURE	Pointer to function that configures the MDIX mode for the Ethernet PHY.
	DRV_ETHPHY_VENDOR_MII_CONFIGURE	Pointer to function to configure the Ethernet PHY in one of the MII/RMII operation modes.
	DRV_ETHPHY_VENDOR_SMI_CLOCK_GET	Pointer to a function to return the SMI/MIIM maximum clock speed in Hz of the Ethernet PHY.
	DRV_ETHPHY_VENDOR_WOL_CONFIGURE	Pointer to a function to configure the PHY WOL functionality

Description

Ethernet ETHPHY Device Driver Interface

The Ethernet ETHPHY device driver provides a simple interface to manage an Ethernet ETHPHY peripheral using MIIM (or SMI) interface. This file defines the interface definitions and prototypes for the Ethernet ETHPHY driver.

File Name

drv_ethphy.h

Company

Microchip Technology Inc.

drv_ethphy_config.h

Ethernet PHY driver configuration definitions template.

Macros

	Name	Description
	DRV_ETHPHY_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ETHPHY_INDEX	Ethernet PHY static index selection.
	DRV_ETHPHY_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_ETHPHY_NEG_DONE_TMO	Value of the PHY negotiation complete time out as per IEEE 802.3 spec.

	DRV_ETHPHY_NEG_INIT_TMO	Value of the PHY negotiation initiation time out as per IEEE 802.3 spec.
	DRV_ETHPHY_PERIPHERAL_ID	Defines an override of the peripheral ID.
	DRV_ETHPHY_RESET_CLR_TMO	Value of the PHY Reset self clear time out as per IEEE 802.3 spec.

Description

Ethernet PHY Driver Configuration Definitions for the Template Version

These definitions statically define the driver's mode of operation.

File Name

drv_ethphy_config.h

Company

Microchip Technology Inc.

Graphics Driver Library

This topic describes the Graphics (GFX) Driver Library.

Introduction

The Graphics (GFX) Driver Layer Library is the GFX library stack available for the Microchip family of microcontrollers.

Description

The Microchip Graphics (GFX) Library is a free, modular library optimized for Microchip Microcontrollers.

- Graphics Object Layer - This layer renders the control objects such as button, list box, progress bar, meter, and so on
- Graphics Primitives Layer - This layer implements the primitive rendering functions
- Graphics Display Driver Layer - This layer is the graphics display driver component that is optimized to the actual display module used

The library comes with features such as alpha blending, gradient fills, and anti-aliased fonts. These features can be enabled or disabled through build configurations. Applications can take advantage of these features to enhance the user experience while delivering performance required by the application.

This help provides information on the Graphics Display Driver Layer. This layer is the interface between the Graphics Core Library and the hardware - which may include a display controller or directly to the display device.

Using the Library

This topic describes the basic architecture of the Graphics Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_gfx_display.h`

The interface to the Graphics Driver Library is defined in the `drv_gfx_display.h` header file.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Library Overview

This topic provides and overview of the Graphics Driver Library.

Description

The driver library API is described in three groups.

- Initialization - provides system level initialization support
- Operations - provides application level run-time support
- Rendering - provides core library level display drawing support

System and Application functions are accessed by system and application routines without using the graphics core library. Rendering function (Library Functions) are intended be used by the Graphics Primitive Layer only to render pixel data to the actual display.

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The Library Interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Graphics Driver Library.

Section	Description
Graphics Driver Library Functions	Provides general interface functions specific to the Graphics Driver Library.
LCC Graphics Driver Functions	Provides interface functions specific to the Low-Cost Controllerless (LCC) Graphics Driver Library.
OTM2201A Graphics Driver Functions	Provides interface functions specific to the OTM2201A Graphics Driver Library.
S1D13517 Graphics Driver Functions	Provides interface functions specific to the S1D13517 Graphics Driver Library.
SSD1289 Graphics Driver Functions	Provides interface functions specific to the SSD1289 Graphics Driver Library.
SSD1926 Graphics Driver Functions	Provides interface functions specific to the SSD1926 Graphics Driver Library.
tft002 Graphics Driver Functions	Provides interface functions specific to the tft002 Graphics Driver Library.

Initialization

This topic provides information for initializing the Graphics Driver Library.

Description

The Graphics Library Driver Layer provides application routines to open and configure an instance of the driver.

The application prototypes are generic to all drivers. The user is required to supply the appropriate configuration in

the arguments.

```
DRV_HANDLE DRV_GFX_<driver>_Initialize( const SYS_MODULE_INDEX  drvIndex,
                                        const * SYS_MODULE_INIT drvInit)
```

Operations

This topic provides operations information for the Graphics Driver Library.

Description

The Graphics Library Driver Layer provides application routines to open and configure an instance of the driver.

The application prototypes are generic to all drivers. The user is required to supply the appropriate configuration in the arguments.

```
DRV_HANDLE DRV_GFX_<driver>_Open( const SYS_MODULE_INDEX drvIndex,
                                  const DRV_IO_INTENT ioIntent )
```

Rendering

This topic provides information on rendering functions.

Description

The Graphics library driver layer provides basic rendering routines to draw pixels on a specific display.

More information on these interfaces can be found in the Library Interface section.

Creating a New Graphics Driver

This topic provides information for creating a new graphics driver.

Description

The display driver layer provides a generic programming interface to the Graphics Library code allowing it to draw actual pixels on the display. This interface is adopted by all supported drivers in the graphics library. Each driver is not required to implement all functions. The interface defines the necessary functions required to support the core library.

The display driver API includes the following basic functions defined in the `drv_gfx_display.h` file:

Return type	Function Name	Description
uint16_t	PixelsPut	Sends pixels to the display
uint16_t	BarFill	Renders a bar to the display
uint16_t*	PixelArrayPut	Sends an array of pixels to the display
uint16_t*	PixelArrayGet	Returns an array of pixels from the display
uint16_t	PixelPut	Sends a pixel to the display.
void	ColorSet	Sets the pixel color.
void	InstanceSet	Sets the instance.
uint16_t	PageSet	Sets a page according to the enum PAGE_TYPE.
uint16_t*	Layer	
uint16_t	PixelGet	Returns a pixel.
uint16_t*	AlphaBlendWindow	Alpha window
GFX_STATUS	Status	Returns the status of the driver.

When creating a new driver, the developer will need to provide to the Graphics Library the set of function by means of the function pointer table structure defined in `gfx_driver_display.h`.

The main application code examples poll the DRV_GFX_XXX_InterfaceSet functions to recall which functions are available from a given driver. For faster throughput, DRV_GFX macros can be created (refer to `gfx_driver_display.h` for examples).

Not all rendering functions are needed to for a graphics controller driver to be usable with the library. The inclusion of the functions are dependant on application requirements. For example, if the controller does not have alphablending, the primitive layer can perform this task. When the initial driver structure is created these functions need to be set as NULL. Any of the "put" functions need to be defined by the controller driver.

Configuring the Library

LCC Driver Configuration Functions

	Name	Description
	DRV_GFX_CONFIG_LCC_EXTERNAL_MEMORY	Macro that enables external memory framebuffer.
	DRV_GFX_CONFIG_LCC_INTERNAL_MEMORY	Macro that enables internal memory framebuffer.
	DRV_GFX_CONFIG_LCC_PALETTE	Macro that disables internal palette memory framebuffer.
	DRV_GFX_LCC_DMA_CHANNEL_INDEX	Macro that defines the DMA CHANNEL INDEX.

OTM2201A Driver Configuration Functions

	Name	Description
	GFX_CONFIG_OTM2201A_DRIVER_COUNT	Macro sets the number of instances for the driver.

S1D13517 Driver Configuration Functions

	Name	Description
	GFX_CONFIG_S1D13517_DRIVER_COUNT	Macro sets the number of instances for the driver.

SSD1926 Driver Configuration Functions

	Name	Description
	GFX_CONFIG_SSD1926_DRIVER_COUNT	Macro sets the number of instances for the driver.

tft002 Driver Configuration Functions

	Name	Description
	GFX_CONFIG_TFT002_DRIVER_COUNT	Macro sets the number of instances for the driver.

Description

The configuration of the GFX driver is based on the file `drv_gfx_config.h`.

This header file contains the configuration selection for the Graphics Driver. Based on the selections made, the Graphics Driver may support the selected features. These configuration settings will apply to all instances of the Graphics Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

The configuration of the Graphics Driver layer is mainly dependent on making sure the needed driver has the core driver functions inside of it. The main rendering one being a `PixelPut` type function for the Graphics primitive layer.

Once the Graphics Driver structure (GFX_DRV_DATA) is defined along with its primitive function rendering structure (GFX_DRV_FUNCTIONS) it can be used by the Graphics primitive layer.

LCC Driver Configuration Functions

DRV_GFX_CONFIG_LCC_EXTERNAL_MEMORY Macro

Macro that enables external memory framebuffer.

File

drv_gfx_lcc_config_template.h

C

```
#define DRV_GFX_CONFIG_LCC_EXTERNAL_MEMORY
```

Description

Macro: DRV_GFX_CONFIG_LCC_EXTERNAL_MEMORY

This macro enables the use of off-chip memory for the graphic framebuffer.

To enable external framebuffer, add this macro in the configuration and delete DRV_GFX_CONFIG_LCC_INTERNAL_MEMORY if defined.

Remarks

Mandatory definition if DRV_GFX_CONFIG_LCC_INTERNAL_MEMORY is not set.

DRV_GFX_CONFIG_LCC_INTERNAL_MEMORY Macro

Macro that enables internal memory framebuffer.

File

drv_gfx_lcc_config_template.h

C

```
#define DRV_GFX_CONFIG_LCC_INTERNAL_MEMORY
```

Description

Macro: DRV_GFX_CONFIG_LCC_EXTERNAL_MEMORY

This macro enables the use of on-chip memory for the graphic framebuffer.

To enable internal framebuffer, add this macro in the configuration and delete DRV_GFX_CONFIG_LCC_EXTERNAL_MEMORY if defined.

Remarks

Mandatory definition if DRV_GFX_CONFIG_LCC_EXTERNAL_MEMORY is not set.

DRV_GFX_CONFIG_LCC_PALETTE Macro

Macro that disables internal palette memory framebuffer.

File

drv_gfx_lcc_config_template.h

C

```
#define DRV_GFX_CONFIG_LCC_PALETTE
```

Description

///////////////// COMPILE OPTIONS ///////////////////

****

Macro: DRV_GFX_LCC_PALETTE_DISABLE

This macro disables the use color lookup palette.

To disable palette internal framebuffer, add this macro in the configuration.

Remarks

Optional definition.

DRV_GFX_LCC_DMA_CHANNEL_INDEX Macro

Macro that defines the DMA CHANNEL INDEX.

File

drv_gfx_lcc_config_template.h

C

```
#define DRV_GFX_LCC_DMA_CHANNEL_INDEX
```

Description

Macro: DRV_GFX_LCC_DMA_CHANNEL_INDEX

This macro defines the DMA CHANNEL to be used for LCC display controller operations.

Remarks

Mandatory definition.

S1D13517 Driver Configuration Functions

GFX_CONFIG_S1D13517_DRIVER_COUNT Macro

Macro sets the number of instances for the driver.

File

drv_gfx_s1d13517_config_template.h

C

```
#define GFX_CONFIG_S1D13517_DRIVER_COUNT
```

Description

//////////////// COMPILE OPTIONS //////////////////

```
*******************************************************************************
```

Macro: GFX_CONFIG_S1D13517_DRIVER_COUNT

This macro sets the number of instances the driver will support.

Remarks

None.

SSD1926 Driver Configuration Functions

GFX_CONFIG_SSD1926_DRIVER_COUNT Macro

Macro sets the number of instances for the driver.

File

drv_gfx_ssd1926_config_template.h

C

```
#define GFX_CONFIG_SSD1926_DRIVER_COUNT
```

Description

//////////////// COMPILE OPTIONS //////////////////

****

Macro: GFX_CONFIG_SSD1926_DRIVER_COUNT

This macro sets the number of instances the driver will support.

Remarks

None.

OTM2201A Driver Configuration Functions

GFX_CONFIG_OTM2201A_DRIVER_COUNT Macro

Macro sets the number of instances for the driver.

File

drv_gfx_otm2201a_config_template.h

C

```
#define GFX_CONFIG_OTM2201A_DRIVER_COUNT
```

Description

///////////////// COMPILE OPTIONS ///////////////////

****

Macro: GFX_CONFIG_OTM2201A_DRIVER_COUNT

This macro sets the number of instances the driver will support.

Remarks

None.

tft002 Driver Configuration Functions

GFX_CONFIG_TFT002_DRIVER_COUNT Macro

Macro sets the number of instances for the driver.

File

drv_gfx_tft002_config_template.h

C

```
#define GFX_CONFIG_TFT002_DRIVER_COUNT
```

Description

///////////////// COMPILE OPTIONS ///////////////////

****

Macro: GFX_CONFIG_TFT002_DRIVER_COUNT

This macro sets the number of instances the driver will support.

Remarks

None.

Configuring the Display

Provides display configuration information.

Description

Each Graphics Controller Driver can support different LCD displays. Each display has its own configuration parameters to correctly display the rendered data. Configuration parameters are located in the file `<install-dir>/framework/driver/gfx/controller/display>/drv_gfx_<display>.h`, where `<display>` indicates the display type. The `drv_gfx_<display>.h` file needs to be included and the configuration parameters are to be passed to the driver initialization structure DRV_GFX_INIT.

Building the Library

This section lists the files that are available in the Graphics Driver Library.

Description

The Graphics Driver is part of the core functionality of the Graphics Library. Please refer to the Building the Library topic in the Graphics Library section for build information.

Library Interface

Graphics Driver Library Data Types and Constants

	Name	Description
	DRV_GFX_INIT	Defines the GFX driver initialization data.
	_GFX_GFX_INIT	Defines the GFX driver initialization data.
	DRV_GFX_LCD	Defines the various states that can be achieved by the driver operation.

LCC Driver Data Types and Constants

	Name	Description
	PIP_BUFFER	This is macro PIP_BUFFER.
	DMA_ISR_TASK	This is type DMA_ISR_TASK.
	DRV_GFX_LCC_INDEX_COUNT	Number of valid LCC driver indices.
	DRV_GFX_LCC_DISPLAY_WRITE_BUS_TYPE	
	DRV_GFX_LCC_FB_WRITE_BUS_TYPE	

LCC Driver Functions

	Name	Description
	GFX_PRIM_SetPIPWindow	returns address to the framebuffer.
	DRV_GFX_LCC_PixelPut	outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_LCC_SetColor	Sets the color for the driver instance
	DRV_GFX_LCC_SetPage	DOM-IGNORE-START
	DRV_GFX_LCC_AlphaBlendWindow	DOM-IGNORE-END
	DRV_GFX_LCC_DisplayRefresh	LCD refresh handler
	DRV_GFX_LCC_BarFill	outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_LCC_Close	closes an instance of the graphics controller
	DRV_GFX_LCC_GetBuffer	DOM-IGNORE-END
	DRV_GFX_LCC_Initialize	resets LCD, initializes PMP
	DRV_GFX_LCC_InterfaceSet	Returns the API of the graphics controller
	DRV_GFX_LCC_Open	opens an instance of the graphics controller
	DRV_GFX_LCC_PixelArrayGet	gets an array of pixels of length count starting at *color
	DRV_GFX_LCC_PixelArrayPut	outputs an array of pixels of length count starting at *color
	DRV_GFX_LCC_Tasks	Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer
	DRV_GFX_PaletteSet	returns address to the framebuffer.
	DRV_GFX_LCC_MaxXGet	Returns x extent of the display.
	DRV_GFX_LCC_MaxYGet	Returns y extent of the display.
	DRV_GFX_LCC_FrameBufferAddressSet	Sets address of the framebuffer

OTM2201A Driver Data Types and Constants

	Name	Description
	OTM2201A_TASK	Enumeration for command type.
	DRV_GFX_OTM2201A_COMMAND	Structure for the commands in the driver queue.
	DRV_GFX_OTM2201A_INDEX_COUNT	Number of valid OTM2201A driver indices.

OTM2201A Driver Functions

	Name	Description
	DRV_GFX_OTM2201A_AddressSet	Sets the start GRAM address where pixel data to be written
	DRV_GFX_OTM2201A_BrightnessSet	Sets the brightness of the display backlight.
	DRV_GFX_OTM2201A_Busy	Returns non-zero value if LCD controller is busy (previous drawing operation is not completed).
	DRV_GFX_OTM2201A_ColorSet	Sets the color for the driver instance
	DRV_GFX_OTM2201A_InstanceSet	Sets the instance for the driver
	DRV_GFX_OTM2201A_PixelArrayGet	Gets an array of pixels of length count into an array starting at *color
	DRV_GFX_OTM2201A_PixelArrayPut	Outputs an array of pixels of length count starting at *color
	DRV_GFX_OTM2201A_PixelPut	Outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_OTM2201A_RegGet	Returns graphics controller register value (byte access)
	DRV_GFX_OTM2201A_BarFill	Outputs count number of pixels into the frame buffer from the given x,y coordinate.
	DRV_GFX_OTM2201A_Close	closes an instance of the graphics controller
	DRV_GFX_OTM2201A_Initialize	resets LCD, initializes PMP
	DRV_GFX_OTM2201A_InterfaceSet	Returns the API of the graphics controller
	DRV_GFX_OTM2201A_Open	opens an instance of the graphics controller
	DRV_GFX_OTM2201A_RegSet	Updates graphics controller register value (byte access)
	DRV_GFX_OTM2201A_Tasks	Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer
	DRV_GFX_OTM2201A_MaxXGet	Returns x extent of the display.
	DRV_GFX_OTM2201A_MaxYGet	Returns y extent of the display.

S1D13517 Driver Data Types and Constants

	Name	Description
	LAYER_REGISTERS	This structure is used to describe layerS1D13517_REGisters.
	DRV_GFX_S1D13517_INDEX_COUNT	Number of valid S1D13517 driver indices.

S1D13517 Driver Functions

	Name	Description
	DRV_GFX_S1D13517_AlphaBlendWindow	SEE primitive layer alphablendWindow definition
	DRV_GFX_S1D13517_BrightnessSet	Sets the brightness of the display backlight.
	DRV_GFX_S1D13517_GetReg	returns graphics controllerS1D13517_REGister value (byte access)
	DRV_GFX_S1D13517_Layer	Updates a Layer depending on the layer parameters.
	DRV_GFX_S1D13517_PixelArrayPut	outputs an array of pixels of length count starting at *color
	DRV_GFX_S1D13517_PixelPut	outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_S1D13517_SetColor	Sets the color for the driver instance
	DRV_GFX_S1D13517_SetInstance	Sets the instance for the driver
	DRV_GFX_S1D13517_SetPage	Sets the page of a certain page type
	DRV_GFX_S1D13517_SetReg	updates graphics controllerS1D13517_REGister value (byte access)
	DRV_GFX_S1D13517_BarFill	outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_S1D13517_Close	closes an instance of the graphics controller

	DRV_GFX_S1D13517_Initialize	resets LCD, initializes PMP
	DRV_GFX_S1D13517_InterfaceSet	Returns the API of the graphics controller
	DRV_GFX_S1D13517_Open	opens an instance of the graphics controller
	DRV_GFX_S1D13517_Tasks	Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer
	DRV_GFX_S1D13517_MaxXGet	Returns x extent of the display.
	DRV_GFX_S1D13517_MaxYGet	Returns y extent of the display.

SSD1289 Driver Functions

	Name	Description
	GFX_TCON_SSD1289Init	Initialize the Solomon Systech SSD1289 Timing Controller.

SSD1926 Driver Data Types and Constants

	Name	Description
	DRV_GFX_SSD1926_INDEX_COUNT	Number of valid SSD1926 driver indices.

SSD1926 Driver Functions

	Name	Description
	DRV_GFX_SSD1926_BarFill	Hardware accelerated barfill function
	DRV_GFX_SSD1926_Busy	Returns non-zero if LCD controller is busy (previous drawing operation is not completed).
	DRV_GFX_SSD1926_GetReg	returns graphics controller register value (byte access)
	DRV_GFX_SSD1926_PixelArrayGet	gets an array of pixels of length count starting at *color
	DRV_GFX_SSD1926_PixelArrayPut	outputs an array of pixels of length count starting at *color
	DRV_GFX_SSD1926_PixelPut	outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_SSD1926_SetColor	Sets the color for the driver instance
	DRV_GFX_SSD1926_SetInstance	Sets the instance for the driver
	DRV_GFX_SSD1926_SetReg	updates graphics controller register value (byte access)
	DRV_GFX_SSD1926_Close	closes an instance of the graphics controller
	DRV_GFX_SSD1926_Initialize	resets LCD, initializes PMP
	DRV_GFX_SSD1926_InterfaceSet	Returns the API of the graphics controller
	DRV_GFX_SSD1926_Open	opens an instance of the graphics controller
	DRV_GFX_SSD1926_Status	Provides the current status of the SSD1926 driver module.
	DRV_GFX_SSD1926_Tasks	Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer
	DRV_GFX_SSD1926_MaxXGet	Returns x extent of the display.
	DRV_GFX_SSD1926_MaxYGet	Returns y extent of the display.

tft002 Driver Data Types and Constants

	Name	Description
	DRV_GFX_TFT002_COMMAND	Structure for the commands in the driver queue.
	DRV_GFX_TFT002_INDEX_COUNT	Number of valid TFT002 driver indices.

tft002 Driver Functions

	Name	Description
	DRV_GFX_TFT002_BrightnessSet	Sets the brightness of the display backlight.

	DRV_GFX_TFT002_Busy	Returns non-zero if LCD controller is busy (previous drawing operation is not completed).
	DRV_GFX_TFT002_Close	closes an instance of the graphics controller
	DRV_GFX_TFT002_GetReg	Returns graphics controller register value (byte access)
	DRV_GFX_TFT002_Initialize	resets LCD, initializes PMP
	DRV_GFX_TFT002_InterfaceGet	Returns the API of the graphics controller
	DRV_GFX_TFT002_MaxXGet	Returns x extent of the display.
	DRV_GFX_TFT002_MaxYGet	Returns y extent of the display.
	DRV_GFX_TFT002_Open	opens an instance of the graphics controller
	DRV_GFX_TFT002_PixelArrayGet	Gets an array of pixels of length count starting at *color.
	DRV_GFX_TFT002_PixelArrayPut	Outputs an array of pixels of length count starting at *color
	DRV_GFX_TFT002_PixelPut	Outputs one pixel into the frame buffer at the x,y coordinate given.
	DRV_GFX_TFT002_PixelsPut	Outputs pixels into the frame buffer starting at the x,y coordinate given.
	DRV_GFX_TFT002_SetColor	Sets the color for the driver instance.
	DRV_GFX_TFT002_SetInstance	Sets the instance for the driver
	DRV_GFX_TFT002_SetReg	updates graphics controller register value (byte access)
	DRV_GFX_TFT002_Status	Returns status of the specific module instance of the Driver module.
	DRV_GFX_TFT002_Tasks	Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer

Description

Graphics Driver Library Application Programming Interface (API).

Graphics Driver Library Data Types and Constants

DRV_GFX_INIT Structure

Defines the GFX driver initialization data.

File

gfx_common.h

C

```
typedef struct _GFX_GFX_INIT {
  SYS_MODULE_INIT moduleInit;
  uint16_t orientation;
  uint16_t horizontalResolution;
  uint16_t verticalResolution;
  uint16_t dataWidth;
  uint16_t horizontalPulseWidth;
  uint16_t horizontalBackPorch;
  uint16_t horizontalFrontPorch;
  uint16_t verticalPulseWidth;
  uint16_t verticalBackPorch;
  uint16_t verticalFrontPorch;
  uint8_t logicShift;
  DRV_GFX_LCD LCDType;
  uint8_t colorType;
  void (* TCON_Init)(void);
  GFX_COLOR color;
  volatile uint8_t activePage;
  volatile uint8_t visualPage;
} DRV_GFX_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
uint16_t orientation;	Orientation of the display (given in degrees of 0,90,180,270)
uint16_t horizontalResolution;	Horizontal Resolution of the displayed orientation in Pixels
uint16_t verticalResolution;	Vertical resolution of the displayed orientation in pixels
uint16_t horizontalPulseWidth;	Horizontal Pulse Width of the LCD
uint16_t horizontalBackPorch;	Horizontal BackPorch of the LCD
uint16_t horizontalFrontPorch;	Horizontal FrontPorch of the LCD
uint16_t verticalPulseWidth;	Vertical Pulse width of the LCD
uint16_t verticalBackPorch;	Vertical BackPorch of the LCD
uint16_t verticalFrontPorch;	Vertical FrontPorch of the LCD
uint8_t logicShift;	Rising/Falling edge indicator of the LCD pixel clock
DRV_GFX_LCD LCDType;	LCD type
uint8_t colorType;	color depth (18BPP, 24BPP, 8BPP, palette)
void (* TCON_Init)(void);	Display signals timing control module initialization
GFX_COLOR color;	current Color set for the display driver
volatile uint8_t activePage;	current activepage set for the display driver
volatile uint8_t visualPage;	current visualPage set for the display driver

Description

GFX Driver Initialize Data

This data type defines data required to initialize or reinitialize the GFX driver.

Remarks

Not all initialization features are available on all devices.

DRV_GFX_LCD Enumeration

Defines the various states that can be achieved by the driver operation.

File

gfx_common.h

C

```
typedef enum {
  LCD_TFT = 1,
  LCD_MSTN2,
  LCD_CSTN2
} DRV_GFX_LCD;
```

Members

Members	Description
LCD_TFT = 1	thin-film transistor
LCD_MSTN2	monochrome super-twisted
LCD_CSTN2	color super-twisted

Description

Driver State Machine States

This enumeration defines the various states that can be achieved by the driver operation.

Remarks

None.

LCC Driver Functions

GFX_PRIM_SetPIPWindow Function

returns address to the framebuffer.

File

drv_gfx_lcc.h

C

```
void GFX_PRIM_SetPIPWindow(uint16_t left, uint16_t top, uint16_t hlength, uint16_t vlength,
uint16_t pipx, uint16_t pipy);
```

Description

none.

DRV_GFX_LCC_PixelPut Function

outputs one pixel into the frame buffer at the x,y coordinate given

File

drv_gfx_lcc.h

C

```
uint16_t DRV_GFX_LCC_PixelPut(short x, short y);
```

Returns

1 - call not successful (LCC driver busy) 0 - call successful

Description

none

Parameters

Parameters	Description
x,y	pixel coordinates

Function

uint16_t DRV_GFX_LCC_PixelPut(short x, short y)

DRV_GFX_LCC_SetColor Function

Sets the color for the driver instance

File

drv_gfx_lcc.h

C

```
void DRV_GFX_LCC_SetColor(GFX_COLOR color);
```

Returns

none

Function

void DRV_GFX_LCC_SetColor(uint8_t instance, GFX_COLOR color)

DRV_GFX_LCC_SetPage Function

File

drv_gfx_lcc.h

C

```
uint16_t DRV_GFX_LCC_SetPage(uint8_t pageType, uint8_t page);
```

Description

DOM-IGNORE-START

DRV_GFX_LCC_AlphaBlendWindow Function

File

drv_gfx_lcc.h

C

```
uint16_t* DRV_GFX_LCC_AlphaBlendWindow(GFX_ALPHA_PARAMS* alphaParams, uint16_t width, uint16_t
height, uint8_t alpha);
```

Description

DOM-IGNORE-END

DRV_GFX_LCC_DisplayRefresh Function

LCD refresh handler

File

drv_gfx_lcc.h

C

```
void DRV_GFX_LCC_DisplayRefresh();
```

Returns

none

Description

This routine is called from the timer interrupt, resulting in a complete LCD update.

Function

void DRV_GFX_LCC_DisplayRefresh(void)

DRV_GFX_LCC_BarFill Function

outputs one pixel into the frame buffer at the x,y coordinate given

File

drv_gfx_lcc.h

C

```
uint16_t DRV_GFX_LCC_BarFill(short left, short top, short right, short bottom);
```

Returns

1 - call not successful (LCC driver busy) 0 - call successful

Description

none

Parameters

Parameters	Description
left,top	pixel coordinates
right, bottom	pixel coordinates

Function

uint16_t DRV_GFX_LCC_BarFill(short left, short top, short right, short bottom)

DRV_GFX_LCC_Close Function

closes an instance of the graphics controller

File

drv_gfx_lcc.h

C

```
void DRV_GFX_LCC_Close(DRV_HANDLE handle);
```

Description

none

Function

void DRV_GFX_LCC_Close(DRV_HANDLE handle)

DRV_GFX_LCC_GetBuffer Function

File

drv_gfx_lcc.h

C

```
unsigned short * DRV_GFX_LCC_GetBuffer();
```

Description

DOM-IGNORE-END

DRV_GFX_LCC_Initialize Function

resets LCD, initializes PMP

File

drv_gfx_lcc.h

C

```
SYS_MODULE_OBJ DRV_GFX_LCC_Initialize(const SYS_MODULE_INDEX moduleIndex, const SYS_MODULE_INIT
* const moduleInit);
```

Returns

1 - call not successful (PMP driver busy) 0 - call successful

Description

none

Parameters

Parameters	Description
instance	driver instance

Function

SYS_MODULE_OBJ DRV_GFX_LCC_Initialize(const SYS_MODULE_INDEX moduleIndex,
const SYS_MODULE_INIT * const moduleInit)

DRV_GFX_LCC_InterfaceSet Function

Returns the API of the graphics controller

File

drv_gfx_lcc.h

C

```
void DRV_GFX_LCC_InterfaceSet(DRV_HANDLE handle, DRV_GFX_INTERFACE * interface);
```

Description

none

Function

DRV_GFX_INTEFACE DRV_GFX_LCC_InterfaceGet(DRV_HANDLE handle)

DRV_GFX_LCC_Open Function

opens an instance of the graphics controller

File

drv_gfx_lcc.h

C

```
DRV_HANDLE DRV_GFX_LCC_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Description

none

Function

DRV_GFX_LCC_Open(uint8_t instance)

DRV_GFX_LCC_PixelArrayGet Function

gets an array of pixels of length count starting at *color

File

drv_gfx_lcc.h

C

```
uint16_t* DRV_GFX_LCC_PixelArrayGet(GFX_COLOR * color, short x, short y, uint16_t count);
```

Returns

ignore

Description

none

Parameters

Parameters	Description
instance	driver instance
*color	start of the array
x	x coordinate of the start point.
y	y coordinate of the end point.
count	number of pixels

Function

uint16_t* DRV_GFX_LCC_PixelArrayGet(uint16_t *color, short x, short y, uint16_t count)

DRV_GFX_LCC_PixelArrayPut Function

outputs an array of pixels of length count starting at *color

File

drv_gfx_lcc.h

C

```
void DRV_GFX_LCC_PixelArrayPut(GFX_COLOR * color, short x, short y, uint16_t count, uint16_t
lineCount);
```

Returns

NULL - call not successful (PMP driver busy) !NULL - address of the display driver queue command

Description

none

Parameters

Parameters	Description
*color	start of the array
x	x coordinate of the start point.
y	y coordinate of the end point.
count	number of pixels
lineCount	number of lines

Function

void DRV_GFX_LCC_PixelArrayPut(uint16_t *color, short x, short y, uint16_t count)

DRV_GFX_LCC_Tasks Function

Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer

File

drv_gfx_lcc.h

C

```
void DRV_GFX_LCC_Tasks(SYS_MODULE_OBJ object);
```

Function

void DRV_GFX_LCC_Tasks(void)

DRV_GFX_PaletteSet Function

returns address to the framebuffer.

File

drv_gfx_lcc.h

C

```
uint8_t DRV_GFX_PaletteSet(GFX_COLOR * pPaletteEntry, uint16_t startEntry, uint16_t length);
```

Description

none.

DRV_GFX_LCC_MaxXGet Function

Returns x extent of the display.

File

drv_gfx_lcc.h

C

```
uint16_t DRV_GFX_LCC_MaxXGet();
```

Example

```
Remarks:
```

Function

void DRV_GFX_LCC_MaxXGet()

DRV_GFX_LCC_MaxYGet Function

Returns y extent of the display.

File

drv_gfx_lcc.h

C

```
uint16_t DRV_GFX_LCC_MaxYGet();
```

Example

```
Remarks:
```

Function

void DRV_GFX_LCC_MaxYGet()

DRV_GFX_LCC_FrameBufferAddressSet Function

Sets address of the framebuffer

File

drv_gfx_lcc.h

C

```
uint16_t DRV_GFX_LCC_FrameBufferAddressSet(void * address);
```

Returns

Sets address of the framebuffer

Description

none

Function

DRV_GFX_LCC_FrameBufferAddressSet(void * address)

LCC Driver Data Types and Constants

PIP_BUFFER Macro

File

drv_gfx_lcc.h

C

```
#define PIP_BUFFER (3)
```

Description

This is macro PIP_BUFFER.

DMA_ISR_TASK Enumeration

File

drv_gfx_lcc.h

C

```
typedef enum {
  ACTIVE_PERIOD = 0,
  BLANKING_PERIOD,
  FINISH_LINE,
  OVERFLOW,
  PIP,
  SCROLL
} DMA_ISR_TASK;
```

Description

This is type DMA_ISR_TASK.

DRV_GFX_LCC_INDEX_COUNT Macro

Number of valid LCC driver indices.

File

drv_gfx_lcc.h

C

```
#define DRV_GFX_LCC_INDEX_COUNT DRV_GFX_LCC_NUMBER_OF_MODULES
```

Description

LCC Driver Module Index Count

This constant identifies LCC driver index definitions.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is device-specific.

DRV_GFX_LCC_DISPLAY_WRITE_BUS_TYPE Enumeration

File

drv_gfx_lcc.h

C

```
typedef enum {
  DRV_GFX_LCC_DISPLAY_WRITE_BUS_TYPE_NONE = 0,
  DRV_GFX_LCC_DISPLAY_WRITE_BUS_TYPE_PMP,
  DRV_GFX_LCC_DISPLAY_WRITE_BUS_TYPE_EBI
} DRV_GFX_LCC_DISPLAY_WRITE_BUS_TYPE;
```

Description

LCC Driver Module Index Count

DRV_GFX_LCC_FB_WRITE_BUS_TYPE Enumeration

File

drv_gfx_lcc.h

C

```
typedef enum {
  DRV_GFX_LCC_FB_WRITE_BUS_TYPE_NONE = 0,
  DRV_GFX_LCC_FB_WRITE_BUS_TYPE_PMP,
  DRV_GFX_LCC_FB_WRITE_BUS_TYPE_EBI
} DRV_GFX_LCC_FB_WRITE_BUS_TYPE;
```

Description

LCC Driver Module Index Count

S1D13517 Driver Functions

DRV_GFX_S1D13517_AlphaBlendWindow Function

SEE primitive layer alphablendWindow definition

File

drv_gfx_s1d13517.h

C

```
uint16_t* DRV_GFX_S1D13517_AlphaBlendWindow(GFX_ALPHA_PARAMS* alphaParams, uint16_t width,
uint16_t height, uint8_t alpha);
```

Function

void DRV_GFX_S1D13517_AlphaBlendWindow(uint8_t pageType, uint8_t page)

DRV_GFX_S1D13517_BrightnessSet Function

Sets the brightness of the display backlight.

File

drv_gfx_s1d13517.h

C

```
void DRV_GFX_S1D13517_BrightnessSet(uint8_t instance, uint16_t level);
```

Returns

none

Description

none

Parameters

Parameters	Description
level	Brightness level. Valid values are 0 to 100. • 0: brightness level is zero or display is turned off • 100: brightness level is maximum

Function

uint8_t DRV_GFX_S1D13517_BrightnessSet(uint8_t instance, uint16_t level)

DRV_GFX_S1D13517_GetReg Function

returns graphics controllerS1D13517_REGister value (byte access)

File

drv_gfx_s1d13517.h

C

```
uint8_t DRV_GFX_S1D13517_GetReg(uint8_t index);
```

Returns

0 - when call was passed

Description

none

Function

uint8_t DRV_GFX_S1D13517_GetReg(uint8_t index)

DRV_GFX_S1D13517_Layer Function

Updates a Layer depending on the layer parameters.

File

drv_gfx_s1d13517.h

C

```
uint16_t* DRV_GFX_S1D13517_Layer(uint8_t type, GFX_LAYER_PARAMS* layer);
```

Returns

NULL - call not successful (PMP driver busy) !NULL - address of the display driver queue command

Description

none

Parameters

Parameters	Description
type	layer type
layer	parameters for Layer function call

Function

uint16_t* DRV_GFX_S1D13517_Layer(uint8_t type, GFX_LAYER_PARAMS* layer)

DRV_GFX_S1D13517_PixelArrayPut Function

outputs an array of pixels of length count starting at *color

File

drv_gfx_s1d13517.h

C

```
void DRV_GFX_S1D13517_PixelArrayPut(uint16_t * color, short x, short y, uint16_t count,
uint16_t lineCount);
```

Returns

NULL - call not successful (PMP driver busy) !NULL - address to the number of pixels yet to be serviced

Description

none

Parameters

Parameters	Description
*color	start of the array
x	x coordinate of the start point.
y	y coordinate of the end point.
count	number of pixels
lineCount	number of lines

Function

void DRV_GFX_S1D13517_PixelArrayPut(uint16_t *color, short x, short y, uint16_t count, uint16_t lineCount)

DRV_GFX_S1D13517_PixelPut Function

outputs one pixel into the frame buffer at the x,y coordinate given

File

drv_gfx_s1d13517.h

C

```
uint16_t DRV_GFX_S1D13517_PixelPut(short x, short y);
```

Returns

NULL - call not successful (PMP driver busy) !NULL - address of the display driver queue command

Description

none

Parameters

Parameters	Description
instance	driver instance
color	color to output
x,y	pixel coordinates

Function

uint16_t DRV_GFX_S1D13517_PixelPut(short x, short y)

DRV_GFX_S1D13517_SetColor Function

Sets the color for the driver instance

File

drv_gfx_s1d13517.h

C

```
void DRV_GFX_S1D13517_SetColor(GFX_COLOR color);
```

Returns

none

Function

void DRV_GFX_S1D13517_SetColor(GFX_COLOR color)

DRV_GFX_S1D13517_SetInstance Function

Sets the instance for the driver

File

drv_gfx_s1d13517.h

C

```
void DRV_GFX_S1D13517_SetInstance(uint8_t instance);
```

Returns

none

Function

void DRV_GFX_S1D13517_SetInstance(uint8_t instance)

DRV_GFX_S1D13517_SetPage Function

Sets the page of a certain page type

File

drv_gfx_s1d13517.h

C

```
uint16_t DRV_GFX_S1D13517_SetPage(uint8_t pageType, uint8_t page);
```

Returns

NULL - call not successful (PMP driver busy) !NULL - address of the display driver queue command

Description

none

Parameters

Parameters	Description
instance	driver instance

Function

void DRV_GFX_S1D13517_SetPage(uint8_t pageType, uint8_t page)

DRV_GFX_S1D13517_SetReg Function

updates graphics controllerS1D13517_REGister value (byte access)

File

drv_gfx_s1d13517.h

C

```
uint16_t DRV_GFX_S1D13517_SetReg(uint8_t index, uint8_t value);
```

Returns

1 - call was not passed 0 - call was passed

Description

none

Parameters

Parameters	Description
value	value to write toS1D13517_REGister

Function

uint8_t DRV_GFX_S1D13517_SetReg(uint8_t index, uint8_t value)

DRV_GFX_S1D13517_BarFill Function

outputs one pixel into the frame buffer at the x,y coordinate given

File

drv_gfx_s1d13517.h

C

```
uint16_t DRV_GFX_S1D13517_BarFill(short left, short top, short right, short bottom);
```

Returns

NULL - call not successful (PMP driver busy) !NULL - address of the display driver queue command

Description

none

Parameters

Parameters	Description
left, top	pixel coordinates
right, bottom	pixel coordinates

Function

uint16_t DRV_GFX_S1D13517_BarFill(short left, short top, short right, short bottom)

DRV_GFX_S1D13517_Close Function

closes an instance of the graphics controller

File

drv_gfx_s1d13517.h

C

```
void DRV_GFX_S1D13517_Close(DRV_HANDLE handle);
```

Returns

0 - instance closed 2 - instance doesn't exist 3 - instance already closed

Description

none

Function

DRV_GFX_S1D13517_Close(uint8_t instance)

DRV_GFX_S1D13517_Initialize Function

resets LCD, initializes PMP

File

drv_gfx_s1d13517.h

C

```
SYS_MODULE_OBJ DRV_GFX_S1D13517_Initialize(const SYS_MODULE_INDEX moduleIndex, const
SYS_MODULE_INIT * const moduleInit);
```

Returns

NULL - call not successful (PMP driver busy) !NULL - address of the display driver queue command

Description

none

Parameters

Parameters	Description
instance	driver instance

Function

uint16_t DRV_GFX_S1D13517_Initialize(uint8_t instance)

DRV_GFX_S1D13517_InterfaceSet Function

Returns the API of the graphics controller

File

drv_gfx_s1d13517.h

C

```
void DRV_GFX_S1D13517_InterfaceSet(DRV_HANDLE handle, DRV_GFX_INTERFACE * interface);
```

Description

none

Function

void DRV_GFX_S1D13517_InterfaceSet(DRV_HANDLE handle, DRV_GFX_INTERFACE * interface)

DRV_GFX_S1D13517_Open Function

opens an instance of the graphics controller

File

drv_gfx_s1d13517.h

C

```
DRV_HANDLE DRV_GFX_S1D13517_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

1 - driver not initialized 2 - instance doesn't exist 3 - instance already open instance to driver when successful

Description

none

Function

DRV_GFX_S1D13517_Open(uint8_t instance)

DRV_GFX_S1D13517_Tasks Function

Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer

File

drv_gfx_s1d13517.h

C

```
void DRV_GFX_S1D13517_Tasks(SYS_MODULE_OBJ object);
```

Function

void DRV_GFX_SSD1926_Tasks(void)

DRV_GFX_S1D13517_MaxXGet Function

Returns x extent of the display.

File

drv_gfx_s1d13517.h

C

```
uint16_t DRV_GFX_S1D13517_MaxXGet();
```

Example

```
Remarks:
```

Function

void DRV_GFX_S1D13517_MaxXGet()

DRV_GFX_S1D13517_MaxYGet Function

Returns y extent of the display.

File

drv_gfx_s1d13517.h

C

```
uint16_t DRV_GFX_S1D13517_MaxYGet();
```

Example

```
Remarks:
```

Function

void DRV_GFX_S1D13517_MaxYGet()

S1D13517 Driver Data Types and Constants

LAYER_REGISTERS Structure

File

drv_gfx_s1d13517.h

C

```
typedef struct {
  uint8_t XStart;
  uint8_t XEnd;
  uint8_t YStart0;
  uint8_t YStart1;
  uint8_t YEnd0;
  uint8_t YEnd1;
  uint8_t StartAddress0;
  uint8_t StartAddress1;
  uint8_t StartAddress2;
} LAYER_REGISTERS;
```

Description

This structure is used to describe layerS1D13517_REGisters.

DRV_GFX_S1D13517_INDEX_COUNT Macro

Number of valid S1D13517 driver indices.

File

drv_gfx_s1d13517.h

C

```
#define DRV_GFX_S1D13517_INDEX_COUNT DRV_GFX_S1D13517_NUMBER_OF_MODULES
```

Description

This constant identifies S1D13517 driver index definitions.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is device-specific.

Section

Data Types and Constants

****

****

****

SSD1926 Driver Module Index Count

SSD1289 Driver Functions

GFX_TCON_SSD1289Init Function

Initialize the Solomon Systech SSD1289 Timing Controller.

File

drv_gfx_ssd1289.h

C

```
void GFX_TCON_SSD1289Init();
```

Returns

None.

Description

Initialize the I/Os to implement a bit-banged SPI interface to connect to the Timing Controller through the SPI.

Function

void GfxTconInit(void)

SSD1926 Driver Functions

DRV_GFX_SSD1926_BarFill Function

Hardware accelerated barfill function

File

drv_gfx_ssd1926.h

C

```
uint16_t DRV_GFX_SSD1926_BarFill(short left, short top, short right, short bottom);
```

Returns

1 - call not successful (PMP driver busy) 0 - call successful

Description

see primitive BarFill

Function

uint16_t DRV_GFX_SSD1926_BarFill(short left, short top, short right, short bottom)

DRV_GFX_SSD1926_Busy Function

Returns non-zero if LCD controller is busy (previous drawing operation is not completed).

File

drv_gfx_ssd1926.h

C

```
uint16_t DRV_GFX_SSD1926_Busy(uint8_t instance);
```

Returns

1 - busy 0 - not busy

Description

none

Parameters

Parameters	Description
instance	driver instance

Function

uint16_t DRV_GFX_SSD1926_Busy(uint8_t instance)

DRV_GFX_SSD1926_GetReg Function

returns graphics controller register value (byte access)

File

drv_gfx_ssd1926.h

C

```
uint8_t DRV_GFX_SSD1926_GetReg(uint16_t index, uint8_t * data);
```

Returns

0 - when call was passed

Description

none

Parameters

Parameters	Description
index	register number
*data	array to store data

Function

uint8_t DRV_GFX_SSD1926_GetReg(uint16_t index, uint8_t *data)

DRV_GFX_SSD1926_PixelArrayGet Function

gets an array of pixels of length count starting at *color

File

drv_gfx_ssd1926.h

C

```
uint16_t* DRV_GFX_SSD1926_PixelArrayGet(uint16_t * color, short x, short y, uint16_t count);
```

Returns

NULL - call not successful !NULL - address of the display driver queue command

Description

none

Parameters

Parameters	Description
instance	driver instance
*color	start of the array
x	x coordinate of the start point.
y	y coordinate of the end point.
count	number of pixels

Function

uint16_t DRV_GFX_SSD1926_PixelArrayGet(uint16_t *color, short x, short y, uint16_t count)

DRV_GFX_SSD1926_PixelArrayPut Function

outputs an array of pixels of length count starting at *color

File

drv_gfx_ssd1926.h

C

```
void DRV_GFX_SSD1926_PixelArrayPut(uint16_t * color, short x, short y, uint16_t count, uint16_t
lineCount);
```

Returns

NULL - call not successful !NULL - handle to the number of pixels remaining

Description

none

Parameters

Parameters	Description
*color	start of the array
x	x coordinate of the start point.
y	y coordinate of the end point.
count	number of pixels
lineCount	number of lines

Function

void DRV_GFX_SSD1926_PixelArrayPut(uint16_t *color, short x, short y, uint16_t count, uint16_t lineCount)

DRV_GFX_SSD1926_PixelPut Function

outputs one pixel into the frame buffer at the x,y coordinate given

File

drv_gfx_ssd1926.h

C

```
uint16_t DRV_GFX_SSD1926_PixelPut(short x, short y);
```

Returns

NULL - call not successful !NULL - address of the display driver queue command

Description

none

Parameters

Parameters	Description
x,y	pixel coordinates

Function

uint16_t DRV_GFX_SSD1926_PixelPut(short x, short y)

DRV_GFX_SSD1926_SetColor Function

Sets the color for the driver instance

File

drv_gfx_ssd1926.h

C

```
void DRV_GFX_SSD1926_SetColor(GFX_COLOR color);
```

Returns

none

Function

void DRV_GFX_SSD1926_SetColor(GFX_COLOR color)

DRV_GFX_SSD1926_SetInstance Function

Sets the instance for the driver

File

drv_gfx_ssd1926.h

C

```
void DRV_GFX_SSD1926_SetInstance(uint8_t instance);
```

Returns

none

Function

void DRV_GFX_SSD1926_SetInstance(uint8_t instance)

DRV_GFX_SSD1926_SetReg Function

updates graphics controller register value (byte access)

File

drv_gfx_ssd1926.h

C

```
uint16_t DRV_GFX_SSD1926_SetReg(uint16_t index, uint8_t value);
```

Returns

1 - call was not passed 0 - call was passed

Description

none

Parameters

Parameters	Description
index	register number
value	value to write to register

Function

uint8_t DRV_GFX_SSD1926_SetReg(uint16_t index, uint8_t value)

DRV_GFX_SSD1926_Close Function

closes an instance of the graphics controller

File

drv_gfx_ssd1926.h

C

```
void DRV_GFX_SSD1926_Close(DRV_HANDLE handle);
```

Description

none

Function

void DRV_GFX_SSD1926_Close(DRV_HANDLE handle)

DRV_GFX_SSD1926_Initialize Function

resets LCD, initializes PMP

File

drv_gfx_ssd1926.h

C

```
SYS_MODULE_OBJ DRV_GFX_SSD1926_Initialize(const SYS_MODULE_INDEX moduleIndex, const
SYS_MODULE_INIT * const moduleInit);
```

Returns

1 - call not successful (PMP driver busy) 0 - call successful

Description

none

Parameters

Parameters	Description
instance	driver instance

Function

SYS_MODULE_OBJ DRV_GFX_SSD1926_Initialize(const SYS_MODULE_INDEX moduleIndex,
const SYS_MODULE_INIT * const moduleInit)

DRV_GFX_SSD1926_InterfaceSet Function

Returns the API of the graphics controller

File

drv_gfx_ssd1926.h

C

```
void DRV_GFX_SSD1926_InterfaceSet(DRV_HANDLE handle, DRV_GFX_INTERFACE * interface);
```

Description

none

Function

void DRV_GFX_SSD1926_InterfaceSet(DRV_HANDLE handle, DRV_GFX_INTERFACE * interface)

DRV_GFX_SSD1926_Open Function

opens an instance of the graphics controller

File

drv_gfx_ssd1926.h

C

```
DRV_HANDLE DRV_GFX_SSD1926_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Description

none

Function

DRV_GFX_SSD1926_Open(uint8_t instance)

DRV_GFX_SSD1926_Status Function

Provides the current status of the SSD1926 driver module.

File

drv_gfx_ssd1926.h

C

```
SYS_STATUS DRV_GFX_SSD1926_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another

Description

This function provides the current status of the SSD1926 driver module.

Preconditions

The DRV_GFX_SSD1926_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_GFX_SSD1926_Initialize
SYS_STATUS          status;

status = DRV_GFX_SSD1926_Status( object );
if( SYS_STATUS_READY != status )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_GFX_SSD1926_Initialize

Function

SYS_STATUS DRV_GFX_SSD1926_Status (SYS_MODULE_OBJ object)

DRV_GFX_SSD1926_Tasks Function

Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer

File

drv_gfx_ssd1926.h

C

```
void DRV_GFX_SSD1926_Tasks(SYS_MODULE_OBJ object);
```

Function

void DRV_GFX_SSD1926_Tasks(void)

DRV_GFX_SSD1926_MaxXGet Function

Returns x extent of the display.

File

drv_gfx_ssd1926.h

C

```
uint16_t DRV_GFX_SSD1926_MaxXGet();
```

Example

```
Remarks:
```

Function

void DRV_GFX_SSD1926_MaxXGet()

DRV_GFX_SSD1926_MaxYGet Function

Returns y extent of the display.

File

drv_gfx_ssd1926.h

C

```
uint16_t DRV_GFX_SSD1926_MaxYGet();
```

Example

```
Remarks:
```

Function

void GFX_MaxYGet()

SSD1926 Driver Data Types and Constants

DRV_GFX_SSD1926_INDEX_COUNT Macro

Number of valid SSD1926 driver indices.

File

drv_gfx_ssd1926.h

C

```
#define DRV_GFX_SSD1926_INDEX_COUNT DRV_GFX_SSD1926_NUMBER_OF_MODULES
```

Description

SSD1926 Driver Module Index Count

This constant identifies SSD1926 driver index definitions.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is device-specific.

OTM2201A Driver Functions

DRV_GFX_OTM2201A_AddressSet Function

Sets the start GRAM address where pixel data to be written

File

drv_gfx_otm2201a.h

C

```
uint16_t DRV_GFX_OTM2201A_AddressSet(uint32_t address);
```

Returns

DRV_OTM2201A_ERROR_PMP_WRITE - returns error during PMP Write, DRV_OTM2201A_ERROR_NO_ERROR - returns success without any error.

Description

Address consists of Lower 8 bit at Register REG_RAM_ADDR_LOW and Higher 8 bit at Register REG_RAM_ADDR_HIGH

Parameters

Parameters	Description
address	pixel address

Function

uint16_t DRV_GFX_OTM2201A_AddressSet(uint32_t address)

DRV_GFX_OTM2201A_BrightnessSet Function

Sets the brightness of the display backlight.

File

drv_gfx_otm2201a.h

C

```
void DRV_GFX_OTM2201A_BrightnessSet(uint8_t instance, uint16_t level);
```

Returns

none

Description

Sets the brightness of the display backlight.

Parameters

Parameters	Description
instance	instance of the driver
level	Brightness level. Valid values are 0 to 100. 0 = brightness level is zero or display is turned off. 100 = brightness level is maximum.

Function

uint8_t DRV_GFX_OTM2201A_BrightnessSet(

uint8_t instance,

uint16_t level

)

DRV_GFX_OTM2201A_Busy Function

Returns non-zero value if LCD controller is busy (previous drawing operation is not completed).

File

drv_gfx_otm2201a.h

C

```
uint16_t DRV_GFX_OTM2201A_Busy(uint8_t instance);
```

Returns

DRV_OTM2201A_ERROR_DEVICE_BUSY - Device is busy, DRV_OTM2201A_ERROR_NO_ERROR - Success, driver is not busy.

Description

Returns non-zero value if LCD controller is busy (previous drawing operation is not completed).

Parameters

Parameters	Description
instance	driver instance

Function

uint16_t DRV_GFX_OTM2201A_Busy(uint8_t instance)

DRV_GFX_OTM2201A_ColorSet Function

Sets the color for the driver instance

File

drv_gfx_otm2201a.h

C

```
void DRV_GFX_OTM2201A_ColorSet(GFX_COLOR color);
```

Returns

none

Description

Sets the color for the driver instance

Parameters

Parameters	Description
color	16 bit 565 format color value

Function

void DRV_GFX_OTM2201A_ColorSet(GFX_COLOR color)

DRV_GFX_OTM2201A_InstanceSet Function

Sets the instance for the driver

File

drv_gfx_otm2201a.h

C

```
void DRV_GFX_OTM2201A_InstanceSet(uint8_t instance);
```

Returns

none

Description

Sets the instance for the driver

Parameters

Parameters	Description
instance	driver instance

Function

void DRV_GFX_OTM2201A_InstanceSet(uint8_t instance)

DRV_GFX_OTM2201A_PixelArrayGet Function

Gets an array of pixels of length count into an array starting at *color

File

drv_gfx_otm2201a.h

C

```
uint16_t* DRV_GFX_OTM2201A_PixelArrayGet(uint16_t * color, short x, short y, uint16_t count);
```

Returns

DRV_OTM2201A_ERROR_QUEUE_FULL - OTM2201A command queue is full,
DRV_OTM2201A_ERROR_NO_ERROR - Success without any error.

Description

Gets an array of pixels of length count into an array starting at *color

Parameters

Parameters	Description
color	Pointer to array where color data is to be loaded
x	pixel coordinate on x axis
y	pixel coordinate on y axis
count	count number of pixels

Function

uint16_t DRV_GFX_OTM2201A_PixelArrayGet(uint16_t *color,

short x,

short y,

uint16_t count)

DRV_GFX_OTM2201A_PixelArrayPut Function

Outputs an array of pixels of length count starting at *color

File

drv_gfx_otm2201a.h

C

```
void DRV_GFX_OTM2201A_PixelArrayPut(uint16_t * color, short x, short y, uint16_t count,
uint16_t lineCount);
```

Returns

handle - handle to the number of pixels remaining, DRV_OTM2201A_ERROR_QUEUE_FULL - OTM2201A command queue is full.

Description

Outputs an array of pixels of length count starting at *color

Parameters

Parameters	Description
color	pointer to array of color of pixels
x	pixel coordinate on x axis.
y	pixel coordinate on y axis.
count	count number of pixels
lineCount	lineCount number of display lines

Function

void DRV_GFX_OTM2201A_PixelArrayPut(

uint16_t *color,

short x,

short y,

uint16_t count,

uint16_t lineCount

)

DRV_GFX_OTM2201A_PixelPut Function

Outputs one pixel into the frame buffer at the x,y coordinate given

File

drv_gfx_otm2201a.h

C

```
uint16_t DRV_GFX_OTM2201A_PixelPut(short x, short y);
```

Returns

DRV_OTM2201A_ERROR_QUEUE_FULL - OTM2201A command queue is full,
DRV_OTM2201A_ERROR_NO_ERROR - Success without any error.

Description

Outputs one pixel into the frame buffer at the x,y coordinate given

Parameters

Parameters	Description
x	pixel coordinate on x axis
y	pixel coordinate on y axis

Function

uint16_t DRV_GFX_OTM2201A_PixelPut(short x, short y)

DRV_GFX_OTM2201A_RegGet Function

Returns graphics controller register value (byte access)

File

drv_gfx_otm2201a.h

C

```
uint8_t DRV_GFX_OTM2201A_RegGet(uint16_t index, uint16_t * data);
```

Returns

DRV_OTM2201A_ERROR_PMP_WRITE - returns error during PMP Write, DRV_OTM2201A_ERROR_PMP_READ - returns error during PMP Read, DRV_OTM2201A_ERROR_NO_ERROR - returns success without any error.

Description

Returns graphics controller register value (byte access)

Parameters

Parameters	Description
index	register number
*data	array to store register data

Function

uint8_t DRV_GFX_OTM2201A_RegGet(

uint16_t index,

uint8_t *data

)

DRV_GFX_OTM2201A_BarFill Function

Outputs count number of pixels into the frame buffer from the given x,y coordinate.

File

drv_gfx_otm2201a.h

C

```
uint16_t DRV_GFX_OTM2201A_BarFill(short left, short top, short right, short bottom);
```

Returns

DRV_OTM2201A_ERROR_QUEUE_FULL - OTM2201A command queue is full,
DRV_OTM2201A_ERROR_NO_ERROR - Success without any error.

Description

Outputs count number of pixels into the frame buffer from the given x,y coordinate.

Parameters

Parameters	Description
left	pixel coordinate on x axis
top	pixel coordinate on y axis
right	pixel coordinate on x axis
bottom	pixel coordinate on y axis

Function

uint16_t DRV_GFX_OTM2201A_BarFill(

short left,

short top,

short right,

short bottom

)

DRV_GFX_OTM2201A_Close Function

closes an instance of the graphics controller

File

drv_gfx_otm2201a.h

C

```
void DRV_GFX_OTM2201A_Close(DRV_HANDLE handle);
```

Returns

0 - instance closed 2 - instance doesn't exist 3 - instance already closed

Description

none

Function

DRV_GFX_OTM2201A_Close(uint8_t instance)

DRV_GFX_OTM2201A_Initialize Function

resets LCD, initializes PMP

File

drv_gfx_otm2201a.h

C

```
SYS_MODULE_OBJ DRV_GFX_OTM2201A_Initialize(const SYS_MODULE_INDEX moduleIndex, const
SYS_MODULE_INIT * const moduleInit);
```

Returns

NULL - call not successful (PMP driver busy) !NULL - address of the display driver queue command

Description

none

Parameters

Parameters	Description
instance	driver instance

Function

uint16_t DRV_GFX_OTM2201A_Initialize(uint8_t instance)

DRV_GFX_OTM2201A_InterfaceSet Function

Returns the API of the graphics controller

File

drv_gfx_otm2201a.h

C

```
void DRV_GFX_OTM2201A_InterfaceSet(DRV_HANDLE handle, DRV_GFX_INTERFACE * interface);
```

Description

none

Function

void DRV_GFX_OTM2201A_InterfaceSet(DRV_HANDLE handle, DRV_GFX_INTERFACE * interface)

DRV_GFX_OTM2201A_Open Function

opens an instance of the graphics controller

File

drv_gfx_otm2201a.h

C

```
DRV_HANDLE DRV_GFX_OTM2201A_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

1 - driver not initialized 2 - instance doesn't exist 3 - instance already open instance to driver when successful

Description

none

Function

DRV_GFX_OTM2201A_Open(uint8_t instance)

DRV_GFX_OTM2201A_RegSet Function

Updates graphics controller register value (byte access)

File

drv_gfx_otm2201a.h

C

```
uint16_t DRV_GFX_OTM2201A_RegSet(uint16_t index, uint16_t value);
```

Returns

DRV_OTM2201A_ERROR_PMP_WRITE - returns error during PMP Write, DRV_OTM2201A_ERROR_NO_ERROR - returns success without any error.

Description

This call can set "value" of the register accessed by its "index" and can repeat the same by number of times mentioned in "repeatCount"

Parameters

Parameters	Description
index	register number
value	value to write to register
repeatCount	repeatCount number of times value is to be written to the register.

Function

uint8_t DRV_GFX_OTM2201A_RegSet(

uint16_t index,

uint8_t value,

uint32_t repeatCount

)

DRV_GFX_OTM2201A_Tasks Function

Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer

File

drv_gfx_otm2201a.h

C

```
void DRV_GFX_OTM2201A_Tasks(SYS_MODULE_OBJ object);
```

Function

void DRV_GFX_OTM2201A_Tasks(void)

DRV_GFX_OTM2201A_MaxXGet Function

Returns x extent of the display.

File

drv_gfx_otm2201a.h

C

```
uint16_t DRV_GFX_OTM2201A_MaxXGet();
```

Example

```
Remarks:
```

Function

void DRV_GFX_OTM2201A_MaxXGet()

DRV_GFX_OTM2201A_MaxYGet Function

Returns y extent of the display.

File

drv_gfx_otm2201a.h

C

```
uint16_t DRV_GFX_OTM2201A_MaxYGet();
```

Example

```
Remarks:
```

Function

void DRV_GFX_OTM2201A_MaxYGet()

OTM2201A Driver Data Types and Constants

OTM2201A_TASK Enumeration

Enumeration for command type.

File

drv_gfx_otm2201a.h

C

```
typedef enum {
  INITIALIZE = 0,
  BUSY,
  PUT_ARRAY,
  PUT_PIXELS
} OTM2201A_TASK;
```

Description

Enum: OTM2201A_TASK

Enumeration for command type.

Parameters

Parameters	Description
INITIALIZE	driver initialization task
BUSY	driver busy task
PUT_ARRAY	driver put array task
PUT_PIXELS	driver put pixels task

DRV_GFX_OTM2201A_COMMAND Structure

Structure for the commands in the driver queue.

File

drv_gfx_otm2201a.h

C

```
typedef struct {
  uint8_t instance;
  uint32_t address;
  uint16_t * array;
  uint16_t data;
  uint16_t count;
  uint16_t lineCount;
  OTM2201A_TASK task;
} DRV_GFX_OTM2201A_COMMAND;
```

Description

Structure: DRV_GFX_OTM2201A_COMMAND

Structure for the commands in the driver queue.

Parameters

Parameters	Description
instance	instance of the driver
address	pixel address
array	pointer to array of pixel data
data	pixel color
count	count number of pixels in one line
lineCount	lineCount number of lines of display
task	Type of task (OTM2201A_TASK enum)

DRV_GFX_OTM2201A_INDEX_COUNT Macro

Number of valid OTM2201A driver indices.

File

drv_gfx_otm2201a.h

C

```
#define DRV_GFX_OTM2201A_INDEX_COUNT DRV_GFX_OTM2201A_NUMBER_OF_MODULES
```

Description

This constant identifies OTM2201A driver index definitions.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is device-specific.

Section

Data Types and Constants

****

****

****

OTM2201A Driver Module Index Count

tft002 Driver Functions

DRV_GFX_TFT002_BrightnessSet Function

Sets the brightness of the display backlight.

File

drv_gfx_tft002.h

C

```
void DRV_GFX_TFT002_BrightnessSet(uint8_t instance, uint16_t level);
```

Description

Sets the brightness of the display backlight to the level given by level variable.

Parameters

Parameters	Description
level	Brightness level. Valid values are 0 to 100. • 0: brightness level is zero or display is turned off • 100: brightness level is maximum

Function

uint8_t DRV_GFX_TFT002_BrightnessSet(uint8_t instance, uint16_t level)

DRV_GFX_TFT002_Busy Function

Returns non-zero if LCD controller is busy (previous drawing operation is not completed).

File

drv_gfx_tft002.h

C

```
uint16_t DRV_GFX_TFT002_Busy(uint8_t instance);
```

Returns

1 - busy 0 - not busy

Description

Returns non-zero if LCD controller is busy (previous drawing operation is not completed).

Parameters

Parameters	Description
instance	driver instance

Function

uint16_t DRV_GFX_TFT002_Busy(uint8_t instance)

DRV_GFX_TFT002_Close Function

closes an instance of the graphics controller

File

drv_gfx_tft002.h

C

```
void DRV_GFX_TFT002_Close(DRV_HANDLE handle);
```

Description

Closes the tft002 driver instance, handle of which is given by handle variable.

Function

void DRV_GFX_TFT002_Close(DRV_HANDLE handle)

DRV_GFX_TFT002_GetReg Function

Returns graphics controller register value (byte access)

File

drv_gfx_tft002.h

C

```
uint8_t DRV_GFX_TFT002_GetReg(uint16_t index, uint8_t * data);
```

Returns

0 - when call was passed

Description

Returns graphics controller register value (byte access).

Parameters

Parameters	Description
index	register number
*data	array to store data

Function

uint8_t DRV_GFX_TFT002_GetReg(uint16_t index, uint8_t *data)

DRV_GFX_TFT002_Initialize Function

resets LCD, initializes PMP

File

drv_gfx_tft002.h

C

```
SYS_MODULE_OBJ DRV_GFX_TFT002_Initialize(const SYS_MODULE_INDEX moduleIndex, const
SYS_MODULE_INIT * const moduleInit);
```

Returns

1 - call not successful (PMP driver busy) 0 - call successful

Description

Initializes driver instance having index moduleIndex. Initialization parameters are set by moduleInit structure. It also calls initialization routines of LCD and PMP modules.

Parameters

Parameters	Description
instance	driver instance

Function

SYS_MODULE_OBJ DRV_GFX_TFT002_Initialize(const SYS_MODULE_INDEX moduleIndex,

const SYS_MODULE_INIT * const moduleInit)

DRV_GFX_TFT002_InterfaceGet Function

Returns the API of the graphics controller

File

drv_gfx_tft002.h

C

```
DRV_GFX_INTERFACE * DRV_GFX_TFT002_InterfaceGet(DRV_HANDLE handle);
```

Returns

Returns the driver interfaces to be called by graphics library.

Description

It returns the driver interfaces to be called by graphics library.

Function

DRV_GFX_INTEFACE DRV_GFX_TFT002_InterfaceGet(DRV_HANDLE handle)

DRV_GFX_TFT002_MaxXGet Function

Returns x extent of the display.

File

drv_gfx_tft002.h

C

```
uint16_t DRV_GFX_TFT002_MaxXGet(DRV_HANDLE handle);
```

Description

Returns x extent of the display.

Function

void DRV_GFX_TFT002_MaxXGet()

DRV_GFX_TFT002_MaxYGet Function

Returns y extent of the display.

File

drv_gfx_tft002.h

C

```
uint16_t DRV_GFX_TFT002_MaxYGet(DRV_HANDLE handle);
```

Description

Returns y extent of the display.

Function

void GFX_MaxYGet()

DRV_GFX_TFT002_Open Function

opens an instance of the graphics controller

File

drv_gfx_tft002.h

C

```
DRV_HANDLE DRV_GFX_TFT002_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

Returns the handle of the driver instance.

Description

Opens the tft002 driver instance with index given by index parameter.

Function

DRV_GFX_TFT002_Open(uint8_t instance)

DRV_GFX_TFT002_PixelArrayGet Function

Gets an array of pixels of length count starting at *color.

File

drv_gfx_tft002.h

C

```
uint16_t* DRV_GFX_TFT002_PixelArrayGet(uint16_t * color, short x, short y, uint16_t count);
```

Returns

NULL - call not successful !NULL - address of the display driver queue command

Description

Gets an array of pixels of length count starting at *color.

Parameters

Parameters	Description
instance	driver instance
*color	start of the array
x	x coordinate of the start point.
y	y coordinate of the end point.
count	number of pixels

Function

uint16_t DRV_GFX_TFT002_PixelArrayGet(uint16_t *color, short x, short y, uint16_t count)

DRV_GFX_TFT002_PixelArrayPut Function

Outputs an array of pixels of length count starting at *color

File

drv_gfx_tft002.h

C

```
uint16_t* DRV_GFX_TFT002_PixelArrayPut(uint16_t * color, short x, short y, uint16_t count,
uint16_t lineCount);
```

Returns

NULL - call not successful !NULL - handle to the number of pixels remaining

Description

Outputs an array of pixels of length count starting at *color.

Parameters

Parameters	Description
*color	start of the array
x	x coordinate of the start point.
y	y coordinate of the end point.
count	number of pixels
lineCount	number of lines

Function

uint16_t* DRV_GFX_TFT002_PixelArrayPut(uint16_t *color, short x, short y, uint16_t count, uint16_t lineCount)

DRV_GFX_TFT002_PixelPut Function

Outputs one pixel into the frame buffer at the x,y coordinate given.

File

drv_gfx_tft002.h

C

```
uint16_t DRV_GFX_TFT002_PixelPut(short x, short y);
```

Returns

NULL - call not successful !NULL - address of the display driver queue command

Description

Outputs one pixel into the frame buffer at the x,y coordinate given.

Parameters

Parameters	Description
x,y	pixel coordinates

Function

uint16_t DRV_GFX_TFT002_PixelPut(short x, short y)

DRV_GFX_TFT002_PixelsPut Function

Outputs pixels into the frame buffer starting at the x,y coordinate given.

File

drv_gfx_tft002.h

C

```
uint16_t DRV_GFX_TFT002_PixelsPut(short x, short y, uint16_t count, uint16_t lineCount);
```

Returns

NULL - call not successful !NULL - address of the display driver queue command

Description

Outputs pixesl into the frame buffer starting at the x,y coordinate. Number of pixels are given by count and number of lines is given by lineCount.

Parameters

Parameters	Description
x,y	pixel coordinates
count	• of pixels to put
lineCount	• of lines

Function

uint16_t DRV_GFX_TFT002_PixelsPut(short x, short y, uint16_t count, uint16_t lineCount)

DRV_GFX_TFT002_SetColor Function

Sets the color for the driver instance.

File

drv_gfx_tft002.h

C

```
void DRV_GFX_TFT002_SetColor(GFX_COLOR color);
```

Description

Sets the color for the driver instance.

Function

void DRV_GFX_TFT002_SetColor(GFX_COLOR color)

DRV_GFX_TFT002_SetInstance Function

Sets the instance for the driver

File

drv_gfx_tft002.h

C

```
void DRV_GFX_TFT002_SetInstance(uint8_t instance);
```

Description

Sets the instance of the driver to be referred.

Function

void DRV_GFX_TFT002_SetInstance(uint8_t instance)

DRV_GFX_TFT002_SetReg Function

updates graphics controller register value (byte access)

File

drv_gfx_tft002.h

C

```
uint16_t DRV_GFX_TFT002_SetReg(uint16_t index, uint16_t value);
```

Returns

1 - call was not passed 0 - call was passed

Description

This call can set "value" of the register accessed by its "index".

Parameters

Parameters	Description
index	register number
value	value to write to register

Function

uint8_t DRV_GFX_TFT002_SetReg(uint16_t index, uint8_t value)

DRV_GFX_TFT002_Status Function

Returns status of the specific module instance of the Driver module.

File

drv_gfx_tft002.h

C

```
SYS_STATUS DRV_GFX_TFT002_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR Indicates that the specified module is in an error state

Description

This function returns the status of the specific module instance disabling its operation (and any hardware for driver modules).

Preconditions

The DRV_GFX_TFT002_Initialize function should have been called before calling this function.

Parameters

Parameters	Description
object	DRV_GFX_TFT002 object handle, returned from DRV_GFX_TFT002_Initialize

Function

SYS_STATUS DRV_GFX_TFT002_Status (SYS_MODULE_OBJ object)

DRV_GFX_TFT002_Tasks Function

Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer

File

drv_gfx_tft002.h

C

```
void DRV_GFX_TFT002_Tasks(SYS_MODULE_OBJ object);
```

Description

Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer

Parameters

Parameters	Description
object	driver object.

Function

void DRV_GFX_TFT002_Tasks(SYS_MODULE_OBJ object)

tft002 Driver Data Types and Constants

DRV_GFX_TFT002_COMMAND Structure

Structure for the commands in the driver queue.

File

drv_gfx_tft002.h

C

```
typedef struct {
  uint8_t instance;
  uint32_t address;
  uint16_t * array;
  uint16_t data;
  uint16_t count;
  uint16_t lineCount;
  TFT002_TASK task;
} DRV_GFX_TFT002_COMMAND;
```

Description

Structure: DRV_GFX_TFT002_COMMAND

Structure for the commands in the driver queue.

Parameters

Parameters	Description
instance	instance of the driver
address	pixel address
array	pointer to array of pixel data
data	pixel color
count	count number of pixels in one line
lineCount	lineCount number of lines of display
task	Type of task (TFT002_TASK enum)

DRV_GFX_TFT002_INDEX_COUNT Macro

Number of valid TFT002 driver indices.

File

drv_gfx_tft002.h

C

```
#define DRV_GFX_TFT002_INDEX_COUNT DRV_GFX_TFT002_NUMBER_OF_MODULES
```

Description

TFT002 Driver Module Index Count

This constant identifies TFT002 driver index definitions.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is device-specific.

Files

Files

Name	Description
drv_gfx_lcc.h	Interface for the graphics library where the primitives are rendered and sent to the graphics controller either external or internal
drv_gfx_s1d13517.h	Interface for the graphics library where the primitives are rendered and sent to the graphics controller either external or internal
drv_gfx_ssd1289.h	Interface for the graphics library, which initializes the SYSTECH SSD1289 Timing Controller.
drv_gfx_ssd1926.h	Interface for the graphics library where the primitives are rendered and sent to the graphics controller either external or internal
drv_gfx_otm2201a.h	Interface for the graphics library where the primitives are rendered and sent to the graphics controller either external or internal
drv_gfx_tft002.h	Interface for the graphics library where the primitives are rendered and sent to the graphics controller either external or internal
drv_gfx_lcc_config_template.h	This header file template defines the configurations for Graphics Library Driver LCC.
drv_gfx_s1d13517_config_template.h	This header file template defines the configurations for Graphics Library Driver S1D13517.
drv_gfx_ssd1926_config_template.h	This header file template defines the configurations for Graphics Library Driver SSD1926.
drv_gfx_otm2201a_config_template.h	This header file template defines the configurations for Graphics Library Driver OTM2201A.
drv_gfx_tft002_config_template.h	This header file template defines the configurations for Graphics Library Driver TFT002.

Description

This section lists the source and header files used by the Graphics Driver Library.

drv_gfx_lcc.h

Interface for the graphics library where the primitives are rendered and sent to the graphics controller either external or internal

Enumerations

	Name	Description
	DMA_ISR_TASK	This is type DMA_ISR_TASK.
	DRV_GFX_LCC_DISPLAY_WRITE_BUS_TYPE	
	DRV_GFX_LCC_FB_WRITE_BUS_TYPE	

Functions

	Name	Description
	DRV_GFX_LCC_AlphaBlendWindow	DOM-IGNORE-END
	DRV_GFX_LCC_BarFill	outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_LCC_Close	closes an instance of the graphics controller
	DRV_GFX_LCC_DisplayRefresh	LCD refresh handler
	DRV_GFX_LCC_FrameBufferAddressSet	Sets address of the framebuffer
	DRV_GFX_LCC_GetBuffer	DOM-IGNORE-END

	DRV_GFX_LCC_Initialize	resets LCD, initializes PMP
	DRV_GFX_LCC_InterfaceSet	Returns the API of the graphics controller
	DRV_GFX_LCC_MaxXGet	Returns x extent of the display.
	DRV_GFX_LCC_MaxYGet	Returns y extent of the display.
	DRV_GFX_LCC_Open	opens an instance of the graphics controller
	DRV_GFX_LCC_PixelArrayGet	gets an array of pixels of length count starting at *color
	DRV_GFX_LCC_PixelArrayPut	outputs an array of pixels of length count starting at *color
	DRV_GFX_LCC_PixelPut	outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_LCC_SetColor	Sets the color for the driver instance
	DRV_GFX_LCC_SetPage	DOM-IGNORE-START
	DRV_GFX_LCC_Tasks	Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer
	DRV_GFX_PaletteSet	returns address to the framebuffer.
	GFX_PRIM_SetPIPWindow	returns address to the framebuffer.

Macros

	Name	Description
	DRV_GFX_LCC_INDEX_COUNT	Number of valid LCC driver indices.
	PIP_BUFFER	This is macro PIP_BUFFER.

Description

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the Low-Cost Controllerless (LCC) Graphics Controller.

File Name

drv_gfx_lcc.h

Company

Microchip Technology Inc.

drv_gfx_s1d13517.h

Interface for the graphics library where the primitives are rendered and sent to the graphics controller either external or internal

Functions

	Name	Description
	DRV_GFX_S1D13517_AlphaBlendWindow	SEE primitive layer alphablendWindow definition
	DRV_GFX_S1D13517_BarFill	outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_S1D13517_BrightnessSet	Sets the brightness of the display backlight.
	DRV_GFX_S1D13517_Close	closes an instance of the graphics controller
	DRV_GFX_S1D13517_GetReg	returns graphics controllerS1D13517_REGister value (byte access)
	DRV_GFX_S1D13517_Initialize	resets LCD, initializes PMP
	DRV_GFX_S1D13517_InterfaceSet	Returns the API of the graphics controller
	DRV_GFX_S1D13517_Layer	Updates a Layer depending on the layer parameters.

	DRV_GFX_S1D13517_MaxXGet	Returns x extent of the display.
	DRV_GFX_S1D13517_MaxYGet	Returns y extent of the display.
	DRV_GFX_S1D13517_Open	opens an instance of the graphics controller
	DRV_GFX_S1D13517_PixelArrayPut	outputs an array of pixels of length count starting at *color
	DRV_GFX_S1D13517_PixelPut	outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_S1D13517_SetColor	Sets the color for the driver instance
	DRV_GFX_S1D13517_SetInstance	Sets the instance for the driver
	DRV_GFX_S1D13517_SetPage	Sets the page of a certain page type
	DRV_GFX_S1D13517_SetReg	updates graphics controllerS1D13517_REGister value (byte access)
	DRV_GFX_S1D13517_Tasks	Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer

Macros

	Name	Description
	DRV_GFX_S1D13517_INDEX_COUNT	Number of valid S1D13517 driver indices.

Structures

	Name	Description
	LAYER_REGISTERS	This structure is used to describe layerS1D13517_REGisters.

Description

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the S1D13517 Graphics Controller.

File Name

drv_gfx_s1d13517.c

Company

Microchip Technology Inc.

drv_gfx_ssd1289.h

Interface for the graphics library, which initializes the SYSTECH SSD1289 Timing Controller.

Functions

	Name	Description
	GFX_TCON_SSD1289Init	Initialize the Solomon Systech SSD1289 Timing Controller.

Description

Module for Microchip Graphics Library

This header file contains the function definition for the interface to the SYSTECH SSD1289 Timing Controller.

File Name

drv_gfx_ssd1289.h

Company

Microchip Technology Inc.

drv_gfx_ssd1926.h

Interface for the graphics library where the primitives are rendered and sent to the graphics controller either external or internal

Functions

	Name	Description
	DRV_GFX_SSD1926_BarFill	Hardware accelerated barfill function
	DRV_GFX_SSD1926_Busy	Returns non-zero if LCD controller is busy (previous drawing operation is not completed).
	DRV_GFX_SSD1926_Close	closes an instance of the graphics controller
	DRV_GFX_SSD1926_GetReg	returns graphics controller register value (byte access)
	DRV_GFX_SSD1926_Initialize	resets LCD, initializes PMP
	DRV_GFX_SSD1926_InterfaceSet	Returns the API of the graphics controller
	DRV_GFX_SSD1926_MaxXGet	Returns x extent of the display.
	DRV_GFX_SSD1926_MaxYGet	Returns y extent of the display.
	DRV_GFX_SSD1926_Open	opens an instance of the graphics controller
	DRV_GFX_SSD1926_PixelArrayGet	gets an array of pixels of length count starting at *color
	DRV_GFX_SSD1926_PixelArrayPut	outputs an array of pixels of length count starting at *color
	DRV_GFX_SSD1926_PixelPut	outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_SSD1926_SetColor	Sets the color for the driver instance
	DRV_GFX_SSD1926_SetInstance	Sets the instance for the driver
	DRV_GFX_SSD1926_SetReg	updates graphics controller register value (byte access)
	DRV_GFX_SSD1926_Status	Provides the current status of the SSD1926 driver module.
	DRV_GFX_SSD1926_Tasks	Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer

Macros

	Name	Description
	DRV_GFX_SSD1926_INDEX_COUNT	Number of valid SSD1926 driver indices.

Structures

	Name	Description
	DRV_GFX_SSD1926_COMMAND	Structure for the commands in the driver queue.

Description

None

File Name

drv_gfx_ssd1926.h

Company

Microchip Technology Inc.

drv_gfx_otm2201a.h

Interface for the graphics library where the primitives are rendered and sent to the graphics controller either external

or internal

Enumerations

	Name	Description
	OTM2201A_TASK	Enumeration for command type.

Functions

	Name	Description
	DRV_GFX_OTM2201A_AddressSet	Sets the start GRAM address where pixel data to be written
	DRV_GFX_OTM2201A_BarFill	Outputs count number of pixels into the frame buffer from the given x,y coordinate.
	DRV_GFX_OTM2201A_BrightnessSet	Sets the brightness of the display backlight.
	DRV_GFX_OTM2201A_Busy	Returns non-zero value if LCD controller is busy (previous drawing operation is not completed).
	DRV_GFX_OTM2201A_Close	closes an instance of the graphics controller
	DRV_GFX_OTM2201A_ColorSet	Sets the color for the driver instance
	DRV_GFX_OTM2201A_Initialize	resets LCD, initializes PMP
	DRV_GFX_OTM2201A_InstanceSet	Sets the instance for the driver
	DRV_GFX_OTM2201A_InterfaceSet	Returns the API of the graphics controller
	DRV_GFX_OTM2201A_MaxXGet	Returns x extent of the display.
	DRV_GFX_OTM2201A_MaxYGet	Returns y extent of the display.
	DRV_GFX_OTM2201A_Open	opens an instance of the graphics controller
	DRV_GFX_OTM2201A_PixelArrayGet	Gets an array of pixels of length count into an array starting at *color
	DRV_GFX_OTM2201A_PixelArrayPut	Outputs an array of pixels of length count starting at *color
	DRV_GFX_OTM2201A_PixelPut	Outputs one pixel into the frame buffer at the x,y coordinate given
	DRV_GFX_OTM2201A_RegGet	Returns graphics controller register value (byte access)
	DRV_GFX_OTM2201A_RegSet	Updates graphics controller register value (byte access)
	DRV_GFX_OTM2201A_Tasks	Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer

Macros

	Name	Description
	DRV_GFX_OTM2201A_INDEX_COUNT	Number of valid OTM2201A driver indices.

Structures

	Name	Description
	DRV_GFX_OTM2201A_COMMAND	Structure for the commands in the driver queue.

Description

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the OTM2201A Graphics Controller.

File Name

drv_gfx_otm2201a.h

Company

Microchip Technology Inc.

drv_gfx_tft002.h

Interface for the graphics library where the primitives are rendered and sent to the graphics controller either external or internal

Functions

	Name	Description
	DRV_GFX_TFT002_BrightnessSet	Sets the brightness of the display backlight.
	DRV_GFX_TFT002_Busy	Returns non-zero if LCD controller is busy (previous drawing operation is not completed).
	DRV_GFX_TFT002_Close	closes an instance of the graphics controller
	DRV_GFX_TFT002_GetReg	Returns graphics controller register value (byte access)
	DRV_GFX_TFT002_Initialize	resets LCD, initializes PMP
	DRV_GFX_TFT002_InterfaceGet	Returns the API of the graphics controller
	DRV_GFX_TFT002_MaxXGet	Returns x extent of the display.
	DRV_GFX_TFT002_MaxYGet	Returns y extent of the display.
	DRV_GFX_TFT002_Open	opens an instance of the graphics controller
	DRV_GFX_TFT002_PixelArrayGet	Gets an array of pixels of length count starting at *color.
	DRV_GFX_TFT002_PixelArrayPut	Outputs an array of pixels of length count starting at *color
	DRV_GFX_TFT002_PixelPut	Outputs one pixel into the frame buffer at the x,y coordinate given.
	DRV_GFX_TFT002_PixelsPut	Outputs pixels into the frame buffer starting at the x,y coordinate given.
	DRV_GFX_TFT002_SetColor	Sets the color for the driver instance.
	DRV_GFX_TFT002_SetInstance	Sets the instance for the driver
	DRV_GFX_TFT002_SetReg	updates graphics controller register value (byte access)
	DRV_GFX_TFT002_Status	Returns status of the specific module instance of the Driver module.
	DRV_GFX_TFT002_Tasks	Task machine that renders the driver calls for the graphics library it must be called periodically to output the contents of its circular buffer

Macros

	Name	Description
	DRV_GFX_TFT002_INDEX_COUNT	Number of valid TFT002 driver indices.

Structures

	Name	Description
	DRV_GFX_TFT002_COMMAND	Structure for the commands in the driver queue.

Description

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the tft002 Graphics Controller.

File Name

drv_gfx_tft002.h

Company

Microchip Technology Inc.

drv_gfx_lcc_config_template.h

This header file template defines the configurations for Graphics Library Driver LCC.

Macros

	Name	Description
	DRV_GFX_CONFIG_LCC_EXTERNAL_MEMORY	Macro that enables external memory framebuffer.
	DRV_GFX_CONFIG_LCC_INTERNAL_MEMORY	Macro that enables internal memory framebuffer.
	DRV_GFX_CONFIG_LCC_PALETTE	Macro that disables internal palette memory framebuffer.
	DRV_GFX_LCC_DMA_CHANNEL_INDEX	Macro that defines the DMA CHANNEL INDEX.

Description

Module for Microchip Graphics Library

File Name

drv_gfx_lcc_config_template.h

Company

Microchip Technology Inc.

drv_gfx_s1d13517_config_template.h

This header file template defines the configurations for Graphics Library Driver S1D13517.

Macros

	Name	Description
	GFX_CONFIG_S1D13517_DRIVER_COUNT	Macro sets the number of instances for the driver.

Description

Module for Microchip Graphics Library

File Name

drv_gfx_s1d13517_config_template.h

Company

Microchip Technology Inc.

drv_gfx_ssd1926_config_template.h

This header file template defines the configurations for Graphics Library Driver SSD1926.

Macros

	Name	Description
	GFX_CONFIG_SSD1926_DRIVER_COUNT	Macro sets the number of instances for the driver.

Description

Module for Microchip Graphics Library

File Name

drv_gfx_ssd1926_config_template.h

Company

Microchip Technology Inc.

drv_gfx_otm2201a_config_template.h

This header file template defines the configurations for Graphics Library Driver OTM2201A.

Macros

	Name	Description
	GFX_CONFIG_OTM2201A_DRIVER_COUNT	Macro sets the number of instances for the driver.

Description

Module for Microchip Graphics Library

File Name

drv_gfx_otm2201A_config_template.h

Company

Microchip Technology Inc.

drv_gfx_tft002_config_template.h

This header file template defines the configurations for Graphics Library Driver TFT002.

Macros

	Name	Description
	GFX_CONFIG_TFT002_DRIVER_COUNT	Macro sets the number of instances for the driver.

Description

Module for Microchip Graphics Library

File Name

drv_gfx_tft002_config_template.h

Company

Microchip Technology Inc.

I2C Driver Library Help

This topic describes the I2C Driver Library.

Introduction

This library provides an interface to manage the data transfer operations using the I2C module on the Microchip family of microcontrollers.

Description

This driver library provides application ready routines to read and write data using the I2C protocol, thus minimizing developer's awareness of the working of the I2C protocol.

- Provides read/write and buffer data transfer models
- Supports interrupt and Polled modes of operation
- Support multi-client and multi-instance operation
- Provides data transfer events
- Supports blocking and non-blocking operation
- Supports baud rate setting

Using the Library

This topic describes the basic architecture of the I2C Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_i2c.h`

The interface to the I2C Driver Library is defined in the `drv_i2c.h` header file. Any C language source (`.c`) file that uses the I2C Driver Library should include `drv_i2c.h`.

Library File: The I2C Driver Library archive (`.a`) file is installed with MPLAB Harmony.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

The I2C Driver Library provides the low-level abstraction of the I2C module on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the I2C Driver Library interface.

Description

The I2C Driver Library features routines to perform two functions, driver maintenance and data transfer:

Driver Maintenance

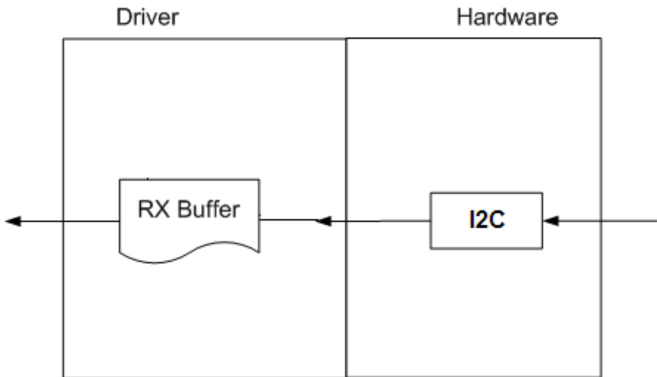
The Driver initialization routines allow the application to initialize the driver. The initialization data configures the I2C module as a Master or a Slave and sets the necessary parameters required for operation in the particular mode. The driver must be initialized before it can be used by the application. After the end of operation, the driver can be deinitialized.

Data Transfer

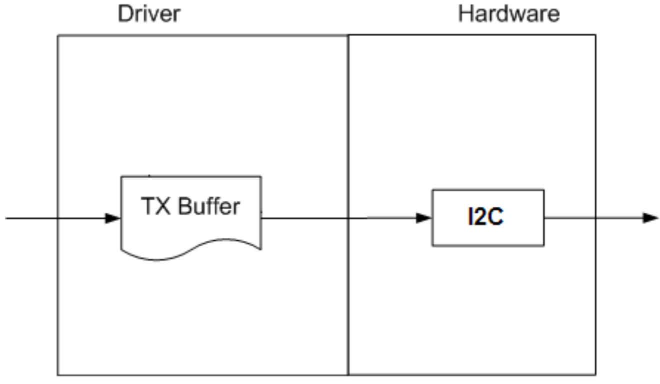
Data transfer is accomplished by separate Write and Read functions through a data buffer. The read and write function makes the user transparent to the internal working of the I2C protocol. The user can use callback mechanisms or use polling to check status of transfer.

The following diagrams illustrate the model used by the I2C Driver for transmitter and receiver.

Receiver Abstraction Model



Transmitter Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the I2C Driver Library.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open, close, status and other setup functions.
Data Transfer Functions	Provides data transfer functions available in the configuration.
Miscellaneous Functions	Provides miscellaneous driver functions.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

System Access

This section provides information on system access.

Description

System Access

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the I2C module would be initialized with the following configuration settings (either passed dynamically at run-time using DRV_I2C_INIT or by using initialization overrides) that are supported by the specific I2C device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- The actual peripheral ID enumerated as the PLIB level module ID (e.g., I2C_ID_2)
- Master or Slave mode of operation and their associated parameters
- Defining the respective interrupt sources for Master, Slave, and Error Interrupt

The DRV_I2C_Initialize API returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the Initialize interface would be used by the other system interfaces like DRV_I2C_Deinitialize, DRV_I2C_Status, and DRV_I2C_Tasks.

Note: The system initialization settings, only affect the instance of the peripheral that is being initialized.

Example:

```
DRV_I2C_INIT        i2c_init_data;
SYS_MODULE_OBJ      objectHandle;

i2c_init_data.i2cId = DRV_I2C_PERIPHERAL_ID_IDX0,
i2c_init_data.i2cMode = DRV_I2C_MODE_MASTER,
OR
i2c_init_data.i2cMode = DRV_I2C_MODE_SLAVE,

/*    Master mode parameters */
i2c_init_data.baudRate = 100000,
i2c_init_data.busspeed = DRV_I2C_SLEW_RATE_CONTROL_IDX0,
i2c_init_data.buslevel = DRV_I2C_SMBus_SPECIFICATION_IDX0,

/*    Master mode parameters */
i2c_init_data.addWidth = DRV_I2C_7BIT_SLAVE,
i2c_init_data.reservedaddenable  = false,
i2c_init_data.generalcalladdress = false,
i2c_init_data.slaveaddvalue = 0x0060,

//interrupt sources
i2c_init_data.mstrInterruptSource = INT_SOURCE_I2C_2_MASTER,
i2c_init_data.slaveInterruptSource = INT_SOURCE_I2C_2_ERROR,
i2c_init_data.errInterruptSource = INT_SOURCE_I2C_2_ERROR,
i2c_init_data.queueSize = 1,

/* callback for Master (Master mode can use callbacks if needed) */
i2c_init_data.operationStarting = NULL,

/* Slave mode callbacks needed */
i2c_init_data.operationStarting = APP_I2CSlaveFunction(),

objectHandle = DRV_I2C_Initialize(DRV_I2C_INDEX_0, (SYS_MODULE_INIT *)&drvI2C0InitData)
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
   // Handle error
}
```

Since the I2C bus is controlled by the Master, the Slave should respond to a read or write request whenever the Master makes the request. Thus, the slave does not have driver states like the Master. The operation of the I2C Driver when used in Slave mode is handled using callbacks. The callback, OperationStarting, must be configured

during system initialization when in Slave mode. This callback is provided so that the application can respond appropriately when a read or write request is received from the Master.

Client Access

This section provides information on client access.

Description

For the application to start using an instance of the module, it must call the DRV_I2C_Open function. This provides the configuration required to open the I2C instance for operation. If the driver is deinitialized using the function DRV_I2C_Deinitialize, the application must call the DRV_I2C_Open function again to set up the instance of the I2C.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

After a client instance is opened, DRV_I2C_ClientSetup can be called to set up client-specific parameters. In I2C Slave mode, this is used to set-up the IRQ logic so that the slave can toggle this line to request Master to send a Read command.

As during initialization, when the I2C module operates in the Slave mode, only the Master can terminate a transaction with the Slave. In this case, the driver provides a callback to the application after the reception of each byte from the Master or after transmission of a byte to the Master.

Example:

```
/* I2C Driver Handle  */
DRV_HANDLE  drvI2CHandle;

/* Open the I2C Driver */
appData.drvI2CHandle = DRV_I2C_Open( DRV_I2C_INDEX_0,DRV_IO_INTENT_WRITE );

if (drvI2CHandle != DRV_HANDLE_VALID)
{
     //Client cannot open instance
}
```

Client Transfer

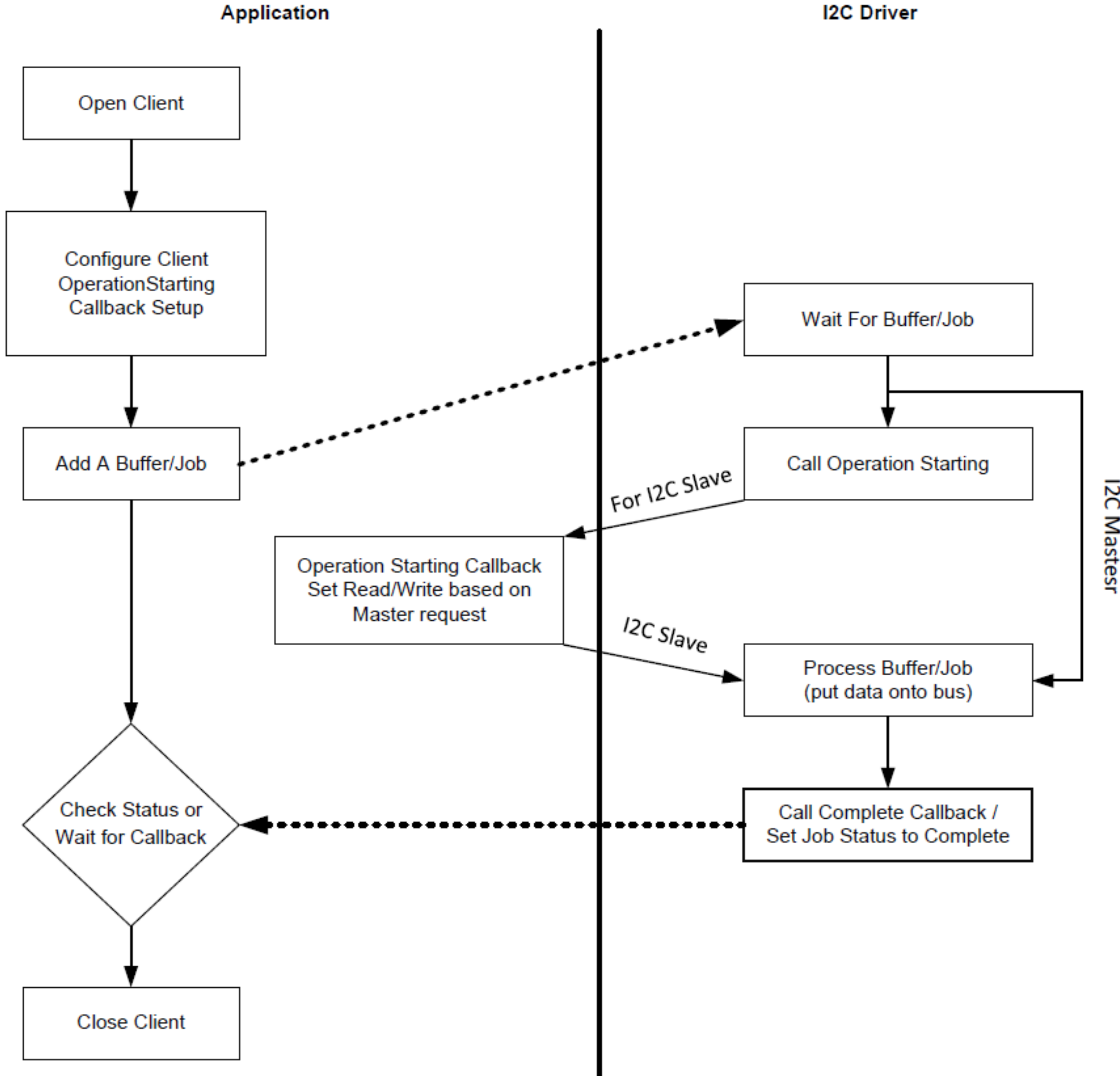
This section provides information on client transfer functionality.

Description

Core Functionality

Client basic functionality provides an extremely basic interface for the driver operation.

The following diagram illustrates the byte/word model used for the data transfer.



Client Data Transfer Functionality

Applications using the I2C driver need to perform the following:

1. The system should have completed necessary initialization and the DRV_I2C_Tasks should either be running in polled environment, or in an interrupt environment.
2. Open the driver using DRV_I2C_Open with the necessary intent.
3. Add a buffer using the DRV_I2C_BufferAddRead/DRV_I2C_BufferAddWrite/DRV_I2C_BufferAddWriteRead functions. An optional callback can be provided that will be called when the buffer/job is complete.

4. Check for the current transfer status using DRV_I2C_BufferStatus until the transfer progress is DRV_I2C_BUFFER_EVENT_COMPLETE, or wait for the callback to be called. If the I2C driver is configured in Polled more, ensure that DRV_I2C_Tasks is called regularly to handle the buffer/job.
5. The client will be able to close the driver using DRV_I2C_Close when required.

Example:

```
SYS_MODULE_OBJ i2cObject;
int main( void )
{
    while ( 1 )
    {
        appTask ();
    }
}
void appTask ()
{
    #define MY_BUFFER_SIZE   5
    DRV_HANDLE    handle;    // Returned from DRV_I2C_Open
    Char    myBuffer[MY_BUFFER_SIZE] = { 11, 22, 33, 44, 55};
    unsigned int  numBytes;
    DRV_I2C_BUFFER_HANDLE bufHandle;

    // Preinitialize myBuffer with MY_BUFFER_SIZE bytes of valid data.
    while( 1 )
    {
        switch( state )
        {
            case APP_STATE_INIT:
                /* Initialize the I2C Driver */
                i2cObject = DRV_I2C_Initialize( DRV_I2C_INDEX_1,
                                              ( SYS_MODULE_INIT * )
                                                &drvI2CInitData );

                /* Check for the System Status */
                if( SYS_STATUS_READY != DRV_I2C_Status( i2cObject ) )
                    return 0;

                /* Open the Driver */
                handle = DRV_I2C_Open( DRV_I2C_INDEX_1,
                                       DRV_IO_INTENT_WRITE );

                /* Update the state to transfer data */
                state = APP_STATE_DATA_PUT;
            break;

            case APP_STATE_DATA_PUT:
                bufHandle = DRV_I2C_BufferAddWrite ( handle, myBuffer,
                                                     5, NULL );
                /* Update the state to status check */
                state = APP_STATE_DATA_CHECK;
            break;

            case APP_STATE_DATA_CHECK:
                /* Check for the successful data transfer */
                if( DRV_I2C_BUFFER_EVENT_COMPLETE &
                    DRV_I2C_BufferStatus( handle ) )
                {
                    /* Do this repeatedly */
                    state = APP_STATE_DATA_PUT;
                }

            break;
            default:
            break;
        }
    }


void __ISR(_I2C_2_VECTOR, ipl4AUTO) _IntHandlerDrvI2CInstance0(void)
```

```
{
    DRV_I2C_Tasks(i2cObject);
}
```

Configuring the Library

Macros

	Name	Description
	DRV_DYNAMIC_BUILD	Dynamic driver build, dynamic device instance parameters.
	DRV_I2C_CONFIG_BUILD_TYPE	Selects static or dynamic driver build configuration.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BASIC	Enables the device driver to support basic transfer mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BLOCKING	Enables the device driver to support blocking operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_EXCLUSIVE	Enables the device driver to support operation in Exclusive mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_MASTER	Enables the device driver to support operation in Master mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_NON_BLOCKING	Enables the device driver to support non-blocking during operations
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_READ	Enables the device driver to support read operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_SLAVE	Enables the device driver to support operation in Slave mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE	Enables the device driver to support write operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE_READ	Enables the device driver to support write followed by read.
	DRV_STATIC_BUILD	Static driver build, static device instance parameters.

Description

The configuration of the I2C Driver Library is based on the file `sys_config.h`.

This header file contains the configuration selection for the I2C Driver Library. Based on the selections made, the I2C Driver Library may support the selected features. These configuration settings will apply to all instances of the I2C Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

DRV_DYNAMIC_BUILD Macro

Dynamic driver build, dynamic device instance parameters.

File

drv_i2c_config_template.h

C

```
#define DRV_DYNAMIC_BUILD 1
```

Description

Dynamic Driver Build Configuration

This value, if used to identify the build type for a driver, will cause the driver to be built to dynamically, identify the instance of the peripheral at run-time using the parameter passed into its API routines.

DRV_I2C_CONFIG_BUILD_TYPE Macro

Selects static or dynamic driver build configuration.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_BUILD_TYPE DRV_DYNAMIC_BUILD
```

Description

I2C Driver Build Configuration Type

This definition selects if I2C device driver is to be used with static or dynamic build parameters. Must be equated to one of the following values:

- DRV_STATIC_BUILD - Build the driver using static accesses to the peripheral identified by the DRV_I2C_INSTANCE macro

- DRV_DYNAMIC_BUILD - Build the driver using dynamic accesses to the peripherals

Affects all the drv_i2c.h driver functions.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BASIC Macro

Enables the device driver to support basic transfer mode.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BASIC
```

Description

Support Basic Transfer Mode

This definition enables the device driver to support basic transfer mode.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_BufferAddReadRead
- DRV_I2C_BufferAddReadWrite

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BLOCKING Macro

Enables the device driver to support blocking operations.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BLOCKING
```

Description

Support Blocking Operations

This definition enables the device driver to support blocking operations.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_Open
- DRV_I2C_Close
- DRV_I2C_BufferAddRead
- DRV_I2C_BufferAddWrite
- DRV_I2C_BufferAddWriteRead

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_EXCLUSIVE Macro

Enables the device driver to support operation in Exclusive mode.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_EXCLUSIVE
```

Description

Support Exclusive Mode

This definition enables the device driver to support operation in Exclusive mode.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_Open

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_MASTER Macro

Enables the device driver to support operation in Master mode.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_MASTER
```

Description

Support Master Mode

This definition enables the device driver to support operation in Master mode.

Remarks

During the configuration phase, the driver selects a list of operation modes that can be supported. While initializing a hardware instance, the device driver will properly perform the initialization base on the selected modes.

The device driver can support multiple modes within a single build.

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_NON_BLOCKING Macro

Enables the device driver to support non-blocking during operations

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_NON_BLOCKING
```

Description

Support Non-Blocking Operations

This definition enables the device driver to support non-blocking operations.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_Open
- DRV_I2C_Close

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_READ Macro

Enables the device driver to support read operations.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_READ
```

Description

Support Read Mode

This definition enables the device driver to support read operations.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_BufferAddRead

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_SLAVE Macro

Enables the device driver to support operation in Slave mode.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_SLAVE
```

Description

Support Slave Mode

This definition enables the device driver to support operation in Slave mode.

Remarks

During the configuration phase, the driver selects a list of operation modes that can be supported. While initializing a hardware instance, the device driver will properly perform the initialization base on the selected modes.

The device driver can support multiple modes within a single build.

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE Macro

Enables the device driver to support write operations.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE
```

Description

Support Write Mode

This definition enables the device driver to support write operations.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_BufferAddWrite

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE_READ Macro

Enables the device driver to support write followed by read.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE_READ
```

Description

Support Write followed by a Read using Restart

This definition enables the device driver to support write followed by read without relinquishing control of the bus. Restart is issued instead of Stop at the end of write. Stop is issued after read operation.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_BufferAddWriteRead

Refer to the description of each function in the corresponding help file for details.

DRV_STATIC_BUILD Macro

Static driver build, static device instance parameters.

File

drv_i2c_config_template.h

C

```
#define DRV_STATIC_BUILD 0
```

Description

Static Driver Build Configuration

This value, if used to identify the build type for a driver, will cause the driver to be built using a specific statically identified instance of the peripheral.

Building the Library

This section lists the files that are available in the I2C Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/i2c`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_i2c.h	This file provides the interface definitions of the I2C driver.
/src/drv_i2c_local.h	This file provides definitions of the data types that are used in the driver object.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_i2c.c	This file contains the core implementation of the I2C driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files exist for this library.

Module Dependencies

The I2C Driver Library depends on the following modules:

- Clock System Service Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_I2C_Deinitialize	Deinitializes the index instance of the I2C module. **Implementation:** Static/Dynamic
	DRV_I2C_Initialize	Initializes hardware and data for the index instance of the I2C module. **Implementation:** Static/Dynamic
	DRV_I2C_Status	Provides the current status of the index instance of the I2C module. **Implementation:** Dynamic
	DRV_I2C_Tasks	Maintains the State Machine of the I2C driver and performs all the protocol level actions. **Implementation:** Dynamic
	DRV_I2C_BaudRateSet	Sets the Baud-Rate for the I2C module. **Implementation:** Static
	DRV_I2C_MasterACKSend	Sends an ACK to the slave. **Implementation:** Static
	DRV_I2C_MasterBusIdle	Checks if the I2C bus is idle. **Implementation:** Static
	DRV_I2C_MasterNACKSend	Sends an ACK to the slave. **Implementation:** Static
	DRV_I2C_MasterStart	Issues a START on I2C bus. **Implementation:** Static
	DRV_I2C_MasterStop	Issues a STOP on I2C bus. **Implementation:** Static
	DRV_I2C_MasterRestart	Issues a START on I2C bus. **Implementation:** Static

b) Client Setup Functions

	Name	Description
	DRV_I2C_Close	Closes an opened instance of an I2C module driver. **Implementation:** Dynamic
	DRV_I2C_Open	Opens the specified instance of the I2C driver for use and provides an "open-instance" handle. **Implementation:** Dynamic

c) Data Transfer Functions

	Name	Description
	DRV_I2C_BufferStatus	Returns status of data transfer when Master or Slave acts either as a transmitter or a receiver. **Implementation:** Dynamic
	DRV_I2C_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_I2C_ByteRead	Reads data from the receive register. **Implementation:** Static
	DRV_I2C_ByteWrite	Writes data byte to the appropriate module. **Implementation:** Static

	DRV_I2C_SetUpByteRead	Prepare Master or Slave for receiving data. **Implementation:** Static
	DRV_I2C_BufferAddRead	This function reads data written from either Master or Slave. **Implementation:** Dynamic
	DRV_I2C_BufferAddWrite	This function writes data to Master or Slave. **Implementation:** Dynamic
	DRV_I2C_BufferAddWriteRead	This function writes data to Slave, inserts restart and requests read from slave. **Implementation:** Dynamic
	DRV_I2C_RestartEventSend	Master sends STOP condition on I2C bus indicating end of WRITE or READ. **Implementation:** Dynamic
	DRV_I2C_StopEventSend	Master sends STOP condition on I2C bus indicating end of WRITE or READ. **Implementation:** Dynamic

d) Status Functions

	Name	Description
	DRV_I2C_WaitForByteWriteToComplete	Checks for the write operation to be completed. **Implementation:** Static
	DRV_I2C_WaitForReadByteAvailable	Checks if data is available in the receive register. **Implementation:** Static
	DRV_I2C_WaitForStartComplete	Checks if a successful START was issued on the I2C bus. **Implementation:** Static
	DRV_I2C_WaitForStopComplete	Checks if a successful STOP was issued on the I2C bus. **Implementation:** Static
	DRV_I2C_WriteByteAcknowledged	Checks if byte written was ACK'ed or NACK'ed. **Implementation:** Static
	DRV_I2C_ReceiverBufferIsEmpty	Checks if no byte is present in Receive register. **Implementation:** Static
	DRV_I2C_WaitForACKOrNACKComplete	Waits for ACK/NACK sequence to be complete. **Implementation:** Static

e) Miscellaneous Functions

	Name	Description
	DRV_I2C_IRQEventSend	IRQ line logic to let Master know that Slave has data to be sent. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	_DRV_I2C_INIT	Identifies the initialization values that are passed as parameters to the initialize and reinitialize routines of the I2C module.
	DRV_I2C_ADDRESS_WIDTH	Lists the Address Width of the Slave.
	DRV_I2C_BUFFER_EVENT	Lists the different conditions that happens during a buffer transfer.
	DRV_I2C_BUFFER_EVENT_HANDLER	Points to a callback after completion of an I2C transfer.
	DRV_I2C_BUFFER_HANDLE	This is type DRV_I2C_BUFFER_HANDLE.
	DRV_I2C_BUS_LEVEL	Lists the Operational Voltage level of I2C.
	DRV_I2C_BUS_SPEED	Lists the I2C Bus speed mode.

	DRV_I2C_CallBack	Points to a callback to initiate a particular function.
	DRV_I2C_INIT	Identifies the initialization values that are passed as parameters to the initialize and reinitialize routines of the I2C module.
	DRV_I2C_INIT_CODE	Codes used for an I2C module initialization.
	DRV_I2C_MODE	Lists the operation mode of I2C module.
	DRV_I2C_SLAVE_ADDRESS_MASK	HV104_0217 Added variable for Slave address Mask
	DRV_I2C_TRANSFER_HANDLE	Handle to an ongoing I2C transfer.
	I2C_DATA_TYPE	This is type I2C_DATA_TYPE.
	I2C_SLAVE_ADDRESS_7bit	need to type cast 10-bit slave address into 8 bit mode
	I2C_SLAVE_ADDRESS_VALUE	This is type I2C_SLAVE_ADDRESS_VALUE.
	DRV_I2C_INDEX_0	The number of I2C instances that are present on the device. This number is dependent on the type of device.
	DRV_I2C_INDEX_1	This is macro DRV_I2C_INDEX_1.
	DRV_I2C_INDEX_2	This is macro DRV_I2C_INDEX_2.
	DRV_I2C_INDEX_3	This is macro DRV_I2C_INDEX_3.
	DRV_I2C_INDEX_4	This is macro DRV_I2C_INDEX_4.
	DRV_I2C_INDEX_5	This is macro DRV_I2C_INDEX_5.
	DRV_I2C_BUFFER_QUEUE_SUPPORT	Specifies if the Buffer Queue support should be enabled.
	DRV_I2C_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_I2C_INTERRUPT_MODE	Macro controls interrupt based operation of the driver
	DRV_I2C_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.

Description

This section describes the Application Programming Interface (API) functions of the I2C Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_I2C_Deinitialize Function

Deinitializes the index instance of the I2C module.

Implementation: Static/Dynamic

File

drv_i2c.h

C

```
void DRV_I2C_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the index instance of the I2C module, disabling its operation (and any hardware for driver modules). It deinitializes only the specified module instance. It also resets all the internal data structures and fields for the specified instance to the default settings.

Remarks

If the module instance has to be used again, DRV_I2C_Initialize should be called again to initialize the module instance structures.

This function may block if the driver is running in an OS environment that supports blocking operations and the driver requires system resources access. However, the routine will NEVER block for hardware I2C access. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_I2C_Status operation. The driver client must always use DRV_I2C_Status to find out when the module is in the ready state.

Preconditions

The DRV_I2C_Initialize function should have been called before calling this function.

Example

```
SYS_STATUS   i2c_status;

DRV_I2C_Deinitialize(I2C_ID_1);

i2c_status = DRV_I2C_Status(I2C_ID_1);
if (SYS_STATUS_BUSY == i2c_status)
{
    // Do something else and check back later
}
else if (SYS_STATUS_ERROR >= i2c_status)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Index, identifying the instance of the I2C module to be deinitialized

Function

void DRV_I2C_Deinitialize (SYS_MODULE_OBJ object)

DRV_I2C_Initialize Function

Initializes hardware and data for the index instance of the I2C module.

Implementation: Static/Dynamic

File

drv_i2c.h

C

```
SYS_MODULE_OBJ DRV_I2C_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const
init);
```

Returns

None.

Description

This function initializes hardware for the index instance of the I2C module, using the hardware initialization given data. It also initializes any internal driver data structures making the driver ready to be opened.

Remarks

This function must be called before any other I2C function is called.

This function should only be called once during system initialization unless DRV_I2C_Deinitialize is first called to deinitialize the device instance before reinitializing it.

This function may block if the driver is running in an OS environment that supports blocking operations and the driver requires system resources access. However, the routine will NEVER block for hardware I2C access. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_I2C_Status operation. The driver client must always use DRV_I2C_Status to find out when the module is in the ready state.

Whenever a call to DRV_I2C_Initialize is made with a SYS_MODULE_INIT* data == 0 the following default configuration will be used. Adjust this configuration at build time as needed.

Preconditions

None.

Example

```
DRV_I2C_INIT        i2c_init_data;
SYS_MODULE_OBJ      objectHandle;

i2c_init_data.i2cId = DRV_I2C_PERIPHERAL_ID_IDX0,
i2c_init_data.i2cMode = DRV_I2C_MODE_MASTER,

OR

i2c_init_data.i2cMode = DRV_I2C_MODE_SLAVE,
//Master mode parameters
i2c_init_data.baudRate = 100000,
i2c_init_data.busspeed = DRV_I2C_SLEW_RATE_CONTROL_IDX0,
i2c_init_data.buslevel = DRV_I2C_SMBus_SPECIFICATION_IDX0,

//Slave mode parameters
i2c_init_data.addWidth = DRV_I2C_7BIT_SLAVE,
i2c_init_data.reservedaddenable  = false,
i2c_init_data.generalcalladdress = false,
i2c_init_data.slaveaddvalue = 0x0060,

//interrupt sources
i2c_init_data.mstrInterruptSource = INT_SOURCE_I2C_2_MASTER,
i2c_init_data.slaveInterruptSource = INT_SOURCE_I2C_2_ERROR,
i2c_init_data.errInterruptSource = INT_SOURCE_I2C_2_ERROR,
i2c_init_data.queueSize = 1,
```

```
//callback for Master (Master mode can use callbacks if needed)
i2c_init_data.operationStarting = NULL,
// Slave mode callbacks needed
i2c_init_data.operationStarting = APP_I2CSlaveFunction(),
i2c_init_data.operationEnded = NULL

objectHandle = DRV_I2C_Initialize(DRV_I2C_INDEX_0, (SYS_MODULE_INIT *)&drvI2C0InitData)
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Index, identifying the instance of the I2C module to be initialized
data	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and the default initialization is to be used.

Function

void DRV_I2C_Initialize (const I2C_MODULE_ID index,

const SYS_MODULE_INIT *const data)

DRV_I2C_Status Function

Provides the current status of the index instance of the I2C module.

Implementation: Dynamic

File

drv_i2c.h

C

```
SYS_STATUS DRV_I2C_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that any previous module operation for the specified I2Cmodule has completed.

- SYS_STATUS_BUSY - Indicates that a previous module operation for the specified I2C module has not yet completed

- SYS_STATUS_ERROR - Indicates that the specified I2C module is in an error state

Description

This function provides the current status of the index instance of the I2C module.

Remarks

The DRV_I2C_Status operation can be used to determine when any of the I2C module level operations has completed. The value returned by the DRV_I2C_Status routine hast to be checked after calling any of the I2C module operations to find out when they have completed.

If the DRV_I2C_Status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the DRV_I2C_Status operation return SYS_STATUS_READY, any previous operations have completed.

The DRV_I2C_Status function will NEVER block.

If the DRV_I2C_Status operation returns an error value, the error may be cleared by calling the DRV_I2C_Initialize operation. If that fails, the DRV_I2C_Deinitialize operation will need to be called, followed by the DRV_I2C_Initialize operation to return to normal operations.

Preconditions

The DRV_I2C_Initialize function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;
SYS_STATUS          i2c_status;

i2c_status = DRV_I2C_Status(object);
if (SYS_STATUS_BUSY == i2c_status)
{
    // Do something else and check back later
}
else if (SYS_STATUS_ERROR >= status)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Index, identifying the instance of the I2C module to get status for.

Function

SYS_STATUS DRV_I2C_Status (SYS_MODULE_OBJ object)

DRV_I2C_Tasks Function

Maintains the State Machine of the I2C driver and performs all the protocol level actions.

Implementation: Dynamic

File

drv_i2c.h

C

```
void DRV_I2C_Tasks(SYS_MODULE_OBJ object);
```

Description

This functions maintains the internal state machine of the I2C driver. This function acts as the I2C Master or Slave ISR. When used in polling mode, this function needs to be called repeatedly to achieve I2C data transfer. This function implements all the protocol level details like setting the START condition, sending the address with with R/W request, writing data to the SFR, checking for acknowledge and setting the STOP condition.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance.

Example

SYS_MODULE_OBJ object;

while (true) { DRV_I2C_Tasks (object);

Function

void DRV_I2C_Tasks (SYS_MODULE_OBJ object)

DRV_I2C_BaudRateSet Function

Sets the Baud-Rate for the I2C module.

Implementation: Static

File

drv_i2c.h

C

```
void DRV_I2C_BaudRateSet(I2C_MODULE_ID i2cid, I2C_BAUD_RATE baudRate);
```

Returns

None.

Description

Sets baud rate for the I2C module.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.
baudRate	Baud-Rate of the I2C module.

Section

Instance I2C Master functions

****

****

Function

void DRV_I2C_BaudRateSet(I2C_MODULE_ID i2cid, I2C_BAUD_RATE baudRate)

DRV_I2C_MasterACKSend Function

Sends an ACK to the slave.

Implementation: Static

File

drv_i2c.h

C

```
void DRV_I2C_MasterACKSend(I2C_MODULE_ID i2cid);
```

Returns

None.

Description

After reading a byte from SLAVE, Master sends an ACK if it expects more data.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

void DRV_I2C_MasterACKSend(I2C_MODULE_ID i2cid)

DRV_I2C_MasterBusIdle Function

Checks if the I2C bus is idle.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_MasterBusIdle(I2C_MODULE_ID i2cid);
```

Returns

- TRUE - if I2C Bus is idle
- FALSE - if I2C Bus is not idle

Description

Checks if the last 5 bytes of the I2CxCON register == 0.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

bool DRV_I2C_MasterBusIdle(I2C_MODULE_ID i2cid)

© 2013-2015 Microchip Technology Inc.

DRV_I2C_MasterNACKSend Function

Sends an ACK to the slave.

Implementation: Static

File

drv_i2c.h

C

```
void DRV_I2C_MasterNACKSend(I2C_MODULE_ID i2cid);
```

Returns

None.

Description

After reading a byte from SLAVE, Master sends an NACK if it doesn't want anymore data from the slave. Slave stops transmission when it receives NACK.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

void DRV_I2C_MasterNACKSend(I2C_MODULE_ID i2cid)

DRV_I2C_MasterStart Function

Issues a START on I2C bus.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_MasterStart(I2C_MODULE_ID i2cid);
```

Returns

- TRUE - if START was issued
- FALSE - if any error condition exists on I2C bus

Description

Checks for any overflow on Receiver or Transmitter side and Bus-Collision. If no error condition is present, then START is issued.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

bool DRV_I2C_MasterStart(I2C_MODULE_ID i2cid)

DRV_I2C_MasterStop Function

Issues a STOP on I2C bus.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_MasterStop(I2C_MODULE_ID i2cid);
```

Returns

- TRUE - if STOP was issued
- FALSE - if any error condition exists on I2C bus

Description

Checks for any error on the bus. If no error condition is present, then STOP is issued.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

bool DRV_I2C_MasterStop(I2C_MODULE_ID i2cid)

DRV_I2C_MasterRestart Function

Issues a START on I2C bus.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_MasterRestart(I2C_MODULE_ID i2cid);
```

Returns

- TRUE - if RESTART was issued
- FALSE - if any error condition exists on I2C bus

Description

Checks for any overflow on Receiver or Transmitter side and Bus-Collision. If no error condition is present, then START is issued.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

bool DRV_I2C_MasterRestart(I2C_MODULE_ID i2cid)

b) Client Setup Functions

DRV_I2C_Close Function

Closes an opened instance of an I2C module driver.

Implementation: Dynamic

File

drv_i2c.h

C

```
void DRV_I2C_Close(DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened instance of an I2C module driver, making the specified handle invalid.

Remarks

After calling This function, the handle passed into drvHandle must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_I2C_Open before the caller may use the driver again.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

Example

```
myI2CHandle = DRV_I2C_Open(I2C_ID_1, DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);
// check the valid handle is obtained and the DRV_I2C_ClientStatus() returns
// that the device driver is ready

// Use the device until it is no longer needed

DRV_I2C_Close(myI2CHandle);
```

Parameters

Parameters	Description
drvHandle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_I2C_Close (const DRV_HANDLE drvHandle)

DRV_I2C_Open Function

Opens the specified instance of the I2C driver for use and provides an "open-instance" handle.

Implementation: Dynamic

File

drv_i2c.h

C

```
DRV_HANDLE DRV_I2C_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a value identifying both the caller and the module instance). If an error occurs, the returned value is DRV_HANDLE_INVALID.

If an error occurs, the return value is DRV_HANDLE_INVALID. An error can occur when the following is true:

- if the number of client objects allocated via DRV_I2C_INSTANCES_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid

Description

This function opens the specified instance of the I2C module for use and provides a handle that is required to use the remaining driver routines.

This function opens a specified instance of the I2C module driver for use by any client module and provides an "open-instance" handle that must be provided to any of the other I2C driver operations to identify the caller and the instance of the I2C driver/hardware module.

Remarks

The handle returned is valid until the DRV_I2C_Close routine is called.

This function may block if the driver is running in an OS environment that supports blocking operations and the driver requires system resources access. Regarding the hardware I2C access the operation will behave as instructed by the DRV_IO_INTENT parameter.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

Example

```
DRV_HANDLE              i2c_handle;

i2c_handle = DRV_I2C_Open(I2C_ID_1, DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);
if (DRV_HANDLE_INVALID == i2c_handle)
{
    // Handle open error
}

// Close the device when it is no longer needed.
DRV_I2C_Close(i2c_handle);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the I2C module to be opened.
intent	Flags parameter identifying the intended usage and behavior of the driver. Multiple flags may be ORed together to specify the intended usage of the device. See the DRV_IO_INTENT definition.

Function

```
DRV_HANDLE DRV_I2C_Open ( const I2C_MODULE_ID index,
const                        DRV_IO_INTENT intent )
```

c) Data Transfer Functions

DRV_I2C_BufferStatus Function

Returns status of data transfer when Master or Slave acts either as a transmitter or a receiver.

Implementation: Dynamic

File

drv_i2c.h

C

```
DRV_I2C_BUFFER_EVENT DRV_I2C_BufferStatus(DRV_I2C_BUFFER_HANDLE bufferHandle);
```

Returns

A DRV_I2C_TRANSFER_STATUS value describing the current status of the transfer.

Description

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event. If the event is DRV_I2C_BUFFER_EVENT_COMPLETE, it means that the data was transferred successfully. If the event is DRV_I2C_BUFFER_EVENT_ERROR, it means that the data was not transferred successfully.

Example

```
if(DRV_I2C_BUFFER_EVENT_COMPLETE & DRV_I2C_BufferStatus (appData.drvI2CWRBUFHandle))
{
        //perform action
        return true;
}
else
{
        //perform action
        return false;
}
```

Parameters

Parameters	Description
object	Instance of I2C object

Function

DRV_I2C_BUFFER_EVENT DRV_I2C_BufferStatus (DRV_I2C_BUFFER_HANDLE bufferHandle)

DRV_I2C_BufferEventHandlerSet Function

Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Implementation: Dynamic

File

drv_i2c.h

C

```
void DRV_I2C_BufferEventHandlerSet(const DRV_HANDLE handle, const DRV_I2C_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls either the DRV_I2C_BufferAddRead or DRV_I2C_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C driver instance.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
DRV_I2C_BUFFER_EVENT operationStatus;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2C_BUFFER_HANDLE appData.drvI2CRDBUFHandle

// appData.drvI2CHandle is the handle returned
// by the DRV_I2C_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2C_BufferEventHandlerSet( appData.drvI2CHandle, APP_I2CBufferEventFunction,
                                 operationStatus );

appData.drvI2CRDBUFHandle = DRV_I2C_BufferAddRead(appData.drvI2CHandle, &mybuffer
                                    MY_BUFFER_SIZE, NULL );

if(DRV_I2C_BUFFER_HANDLE_INVALID == appData.drvI2CRDBUFHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2CBufferEventFunction(DRV_I2C_BUFFER_EVENT event,
        DRV_I2C_BUFFER_HANDLE handle, uintptr_t context)
```

```
{

    switch(event)
    {
        case DRV_I2C_SEND_STOP_EVENT:

        DRV_I2C_StopEventSend(sysObj.drvI2C0);

        break;

        case DRV_I2C_BUFFER_EVENT_COMPLETE:

        //perform appropriate action

        break;

        case DRV_I2C_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_I2C_BufferEventHandlerSet

(

const DRV_HANDLE handle,

const DRV_I2C_EVENT_HANDLER eventHandler,

const uintptr_t context

)

DRV_I2C_ByteRead Function

Reads data from the receive register.

Implementation: Static

File

drv_i2c.h

C

```
uint8_t DRV_I2C_ByteRead(I2C_MODULE_ID i2cid);
```

Returns

Byte that was read from the I2CxRCV register.

Description

Reads data from the receive register.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

uint8_t DRV_I2C_ByteRead(I2C_MODULE_ID i2cid)

DRV_I2C_ByteWrite Function

Writes data byte to the appropriate module.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_ByteWrite(I2C_MODULE_ID i2cid, const uint8_t byte);
```

Returns

TRUE if byte write is successful.

Description

Writes data to the transmit register.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used and data byte.

Function

bool DRV_I2C_ByteWrite(I2C_MODULE_ID i2cid, const uint8_t byte)

DRV_I2C_SetUpByteRead Function

Prepare Master or Slave for receiving data.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_SetUpByteRead(I2C_MODULE_ID i2cid);
```

Returns

Status of the operation.

Description

Sets Receiver Enable for Master. When operating in slave mode, releases the Clock for Slave.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

bool DRV_I2C_SetUpByteRead(I2C_MODULE_ID i2cid)

DRV_I2C_BufferAddRead Function

This function reads data written from either Master or Slave.

Implementation: Dynamic

File

drv_i2c.h

C

```
DRV_I2C_BUFFER_HANDLE DRV_I2C_BufferAddRead(DRV_HANDLE handle, uint8_t* slaveaddress, void *
rxBuffer, size_t size, void * context);
```

Returns

None.

Description

Master calls this function to read data send by Slave. The Slave calls this function to read data send by Master. In case of Master, a START condition is initiated on the I2C bus.

Remarks

After calling This function, the handle passed into drvHandle must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_I2C_Open before the caller may use the driver again.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

Example

```
    drvI2CRDBUFHandle = DRV_I2C_BufferAddRead( appData.drvI2CHandle,
                &deviceaddress,
                rxbuffer, num_of_bytes, NULL );
```

Parameters

Parameters	Description
drvHandle	A valid open-instance handle, returned from the driver's open routine
slaveaddress	Device address of slave. If this API is used in Slave mode, then a dummy value can be used
rxBuffer	This buffer holds data is received
size	The number of bytes that the Master expects to read from Slave. This value can be kept as the MAX BUFFER SIZE for slave. This is because the Master controls when the READ operation is terminated.
context	Not implemented, future expansion

Function

DRV_I2C_BUFFER_HANDLE DRV_I2C_BufferAddRead (DRV_HANDLE handle,

uint8_t* slaveaddress,

void *rxBuffer,

size_t size,

void * context);

DRV_I2C_BufferAddWrite Function

This function writes data to Master or Slave.

Implementation: Dynamic

File

drv_i2c.h

C

```
DRV_I2C_BUFFER_HANDLE DRV_I2C_BufferAddWrite(DRV_HANDLE handle, uint8_t* slaveaddress, void *
txBuffer, size_t size, void * context);
```

Returns

None.

Description

Master calls this function to write data to Slave. The Slave calls this function to write data to Master. In case of Master, a START condition is initiated on the I2C bus

Remarks

After calling This function, the handle passed into drvHandle must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_I2C_Open before the caller may use the driver again.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

Example

```
drvI2CWRBUFHandle = DRV_I2C_BufferAddWrite(appData.drvI2CHandle,
        deviceaddress,
        (I2C_DATA_TYPE *)&appData.drvI2CTXbuffer[0], num_of_bytes, NULL);
```

Parameters

Parameters	Description
drvHandle	A valid open-instance handle, returned from the driver's open routine
slaveaddress	Device address of slave. If this API is used in Slave mode, then a dummy value can be used
txBuffer	Contains data to be transffered
size	The number of bytes that the Master expects to write to Slave. This value can be kept as the MAX BUFFER SIZE for slave. This is because the Master controls when the WRITE operation is terminated.
context	Not implemented, future expansion

Function

DRV_I2C_BUFFER_HANDLE DRV_I2C_BufferAddWrite (DRV_HANDLE handle,

uint8_t* slaveaddress,

void *txBuffer,

size_t size,

void * context);

DRV_I2C_BufferAddWriteRead Function

This function writes data to Slave, inserts restart and requests read from slave.

Implementation: Dynamic

File

drv_i2c.h

C

```
DRV_I2C_BUFFER_HANDLE DRV_I2C_BufferAddWriteRead(DRV_HANDLE handle, uint8_t* slaveaddress, void
* txBuffer, size_t wsize, void * rxBuffer, size_t rsize, void * context);
```

Returns

None.

Description

Master calls this function to send a register address value to the slave and then queries the slave with a read request to read the contents indexed by the register location. The Master sends a restart condition after the initial write before sending the device address with R/W = 1. The restart condition prevents the Master from relinquishing the control of the bus. The slave should not use this function.

Remarks

After calling This function, the handle passed into drvHandle must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_I2C_Open before the caller may use the driver again.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

Example

```
    drvI2CRDBUFHandle = DRV_I2C_BufferAddWriteRead( appData.drvI2CHandle,
                deviceaddress,
                (I2C_DATA_TYPE *)&appData.drvI2CTXbuffer[0], registerbytesize,
                rxbuffer, num_of_bytes, NULL );
```

Parameters

Parameters	Description
drvHandle	A valid open-instance handle, returned from the driver's open routine
slaveaddress	Device address of slave. If this API is used in Slave mode, then a dummy value can be used
txBuffer	Contains data to be transffered
wsize	The number of bytes that the Master expects to write to Slave. This value can be kept as the MAX BUFFER SIZE for slave. This is because the Master controls when the WRITE operation is terminated.
rxBuffer	This buffer holds data that is send back from slave after read operation.
rsize	The number of bytes the Master expects to be read from the slave
context	Not implemented, future expansion

Function

DRV_I2C_BUFFER_HANDLE DRV_I2C_BufferAddWriteRead (DRV_HANDLE handle,

uint8_t* slaveaddress,

void *txBuffer,

size_t wsize,

```
void *rxBuffer,
size_t rsize,
void * context);
```

DRV_I2C_RestartEventSend Function

Master sends STOP condition on I2C bus indicating end of WRITE or READ.

Implementation: Dynamic

File

drv_i2c.h

C

```
void DRV_I2C_RestartEventSend(DRV_HANDLE handle);
```

Returns

None.

Description

Master sends RESTART condition. after completing a WRITE. This is done when the Master does not want to relinquish control of bus but want to start another transaction. This usually happens when the Master wants to send a read data from a specific register in the slave device.

Remarks

None.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

DRV_I2C_BufferAddWriteRead which performs a START but instead of terminating the I2C transaction with a STOP, a RESTART issued on the bus.

Example

```
DRV_I2C_RestartEventSend(appData.drvI2CHandle)
```

Parameters

Parameters	Description
object	Instance of the I2C object

Function

void DRV_I2C_RestartEventSend(DRV_HANDLE handle);

DRV_I2C_StopEventSend Function

Master sends STOP condition on I2C bus indicating end of WRITE or READ.

Implementation: Dynamic

File

drv_i2c.h

C

```
void DRV_I2C_StopEventSend(DRV_HANDLE handle);
```

Returns

None.

Description

Master sends STOP condition after completing a WRITE or READ. This will relinquish the control of I2C bus and another Master is free to take control of the bus.

Remarks

None.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

DRV_I2C_BufferAddWrite/DRV_I2C_BufferAddRead which performs a START and WRITE or READ operation on the bus.

Example

```
DRV_I2C_StopEventSend(appData.drvI2CHandle)
```

Parameters

Parameters	Description
object	Instance of the I2C object

Function

void DRV_I2C_StopEventSend(DRV_HANDLE handle);

d) Status Functions

DRV_I2C_WaitForByteWriteToComplete Function

Checks for the write operation to be completed.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_WaitForByteWriteToComplete(I2C_MODULE_ID i2cid);
```

Returns

TRUE if write operation has completed.

Description

Waits for data write to be completed. Checks status of TBF and TRSTAT to determine if write is completed.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used

Function

bool DRV_I2C_WaitForByteWriteToComplete(I2C_MODULE_ID i2cid)

DRV_I2C_WaitForReadByteAvailable Function

Checks if data is available in the receive register.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_WaitForReadByteAvailable(I2C_MODULE_ID i2cid);
```

Returns

Status of the operation.

Description

Checks if data is available in the receive register, by checking the status of RBF. Returns TRUE if data is available and FALSE if RBF = 0.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

bool DRV_I2C_WaitForReadByteAvailable(I2C_MODULE_ID i2cid)

DRV_I2C_WaitForStartComplete Function

Checks if a successful START was issued on the I2C bus.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_WaitForStartComplete(I2C_MODULE_ID i2cid);
```

Returns

TRUE if START is detected.

Description

Waits for START condition to be present on the bus.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

bool DRV_I2C_WaitForStartComplete(I2C_MODULE_ID i2cid)

DRV_I2C_WaitForStopComplete Function

Checks if a successful STOP was issued on the I2C bus.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_WaitForStopComplete(I2C_MODULE_ID i2cid);
```

Returns

TRUE if STOP is detected.

Description

Waits for STOP condition to be present on the bus.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

bool DRV_I2C_WaitForStopComplete(I2C_MODULE_ID i2cid)

DRV_I2C_WriteByteAcknowledged Function

Checks if byte written was ACK'ed or NACK'ed.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_WriteByteAcknowledged(I2C_MODULE_ID i2cid);
```

Returns

TRUE if ACK'ed; FALSE if NACK'ed.

Description

Checks if byte written was ACK'ed or NACK'ed.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

bool DRV_I2C_WriteByteAcknowledged(I2C_MODULE_ID i2cid)

DRV_I2C_ReceiverBufferIsEmpty Function

Checks if no byte is present in Receive register.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_ReceiverBufferIsEmpty(I2C_MODULE_ID i2cid);
```

Returns

TRUE if empty.

Description

Checks if byte is present in receive register.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

bool DRV_I2C_ReceiverBufferIsEmpty(I2C_MODULE_ID i2cid);

DRV_I2C_WaitForACKOrNACKComplete Function

Waits for ACK/NACK sequence to be complete.

Implementation: Static

File

drv_i2c.h

C

```
bool DRV_I2C_WaitForACKOrNACKComplete(I2C_MODULE_ID i2cid);
```

Returns

TRUE if ACK/NACK sequence is complete.

Description

Master waits for the ACK/NACK to be completed it sends to the slave after it receives a byte from slave. Only after ACK/NACK is complete that Master either sets RCEN or sends a STOP.

Parameters

Parameters	Description
i2cid	Instance of the module ID being used.

Function

bool DRV_I2C_WaitForACKOrNACKComplete(I2C_MODULE_ID i2cid)

e) Miscellaneous Functions

DRV_I2C_IRQEventSend Function

IRQ line logic to let Master know that Slave has data to be sent.

Implementation: Dynamic

File

drv_i2c.h

C

```
void DRV_I2C_IRQEventSend(DRV_HANDLE handle);
```

Returns

None.

Description

The Slave can call this function to toggle the IRQ line to request the Master for a READ (i.e., Slave has data to be send to Master).

Remarks

After calling This function, the handle passed into drvHandle must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_I2C_Open before the caller may use the driver again.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

DRV_I2C_ClientSetup should have been called to configure IRQ port pin

Example

```
DRV_I2C_IRQEventSend(appData.drvI2CHandle);
```

Parameters

Parameters	Description
drvHandle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_I2C_IRQEventSend(DRV_HANDLE handle);

f) Data Types and Constants

DRV_I2C_ADDRESS_WIDTH Enumeration

Lists the Address Width of the Slave.

File

drv_i2c.h

C

```
typedef enum {
  DRV_I2C_7BIT_SLAVE,
  DRV_I2C_10BIT_SLAVE
} DRV_I2C_ADDRESS_WIDTH;
```

Members

Members	Description
DRV_I2C_7BIT_SLAVE	I2C Slave 7 bit
DRV_I2C_10BIT_SLAVE	I2C Slave 10 bit

Description

I2C Slave Address Width

This enumeration lists if the I2C module is configured as a 7-bit Slave or a 10-bit Slave.

Remarks

None.

DRV_I2C_BUFFER_EVENT Enumeration

Lists the different conditions that happens during a buffer transfer.

File

drv_i2c.h

C

```
typedef enum {
  DRV_I2C_BUFFER_EVENT_PENDING,
  DRV_I2C_BUFFER_EVENT_COMPLETE,
  DRV_I2C_BUFFER_EVENT_ERROR,
  DRV_I2C_SEND_STOP_EVENT,
  DRV_I2C_SEND_RESTART_EVENT,
  DRV_I2C_BUFFER_SLAVE_READ_REQUESTED,
  DRV_I2C_BUFFER_SLAVE_WRITE_REQUESTED,
  DRV_I2C_BUFFER_SLAVE_READ_BYTE,
  DRV_I2C_BUFFER_SLAVE_WRITE_BYTE
} DRV_I2C_BUFFER_EVENT;
```

Members

Members	Description
DRV_I2C_BUFFER_EVENT_PENDING	Buffer is pending to get processed
DRV_I2C_BUFFER_EVENT_COMPLETE	All data from or to the buffer was transferred successfully.
DRV_I2C_BUFFER_EVENT_ERROR	There was an error while processing the buffer transfer request.
DRV_I2C_SEND_STOP_EVENT	Send Stop by Master
DRV_I2C_SEND_RESTART_EVENT	Send Restart Event by Master
DRV_I2C_BUFFER_SLAVE_READ_REQUESTED	Master sends data to slave
DRV_I2C_BUFFER_SLAVE_WRITE_REQUESTED	Master requests data from slave
DRV_I2C_BUFFER_SLAVE_READ_BYTE	Slave read byte send by Master
DRV_I2C_BUFFER_SLAVE_WRITE_BYTE	Slave send byte to Master

Description

I2C Buffer Events

This enumeration identifies the different conditions that can happen during a buffer transaction. Callbacks can be made with the appropriate buffer condition passed as a parameter to execute the desired action. The application can also poll the BufferStatus flag to check the status of transfer.

The values act like flags and multiple flags can be set.

Remarks

None.

DRV_I2C_BUFFER_EVENT_HANDLER Type

Points to a callback after completion of an I2C transfer.

File

drv_i2c.h

C

```
typedef void (* DRV_I2C_BUFFER_EVENT_HANDLER)(DRV_I2C_BUFFER_EVENT event, DRV_I2C_BUFFER_HANDLE
bufferHandle, uintptr_t context);
```

Description

I2C Buffer Event Callback

This type identifies the I2C Buffer Event. It allows the client driver to register a callback using DRV_I2C_BUFFER_EVENT_HANDLER. By using this mechanism, the driver client will be notified at the completion of the corresponding transfer.

Remarks

A transfer can be composed of various transfer segments. Once a transfer is completed the driver will call the client registered transfer callback.

The callback could be called from ISR context and should be kept as short as possible. It is meant for signaling and it should not be blocking.

Parameters

Parameters	Description
DRV_I2C_BUFFER_EVENT	Status of I2C transfer
bufferHandle	Handle that identifies that identifies the particular Buffer Object
context	For future implementation

Function

void (*DRV_I2C_BUFFER_EVENT_HANDLER) (DRV_I2C_BUFFER_EVENT event, DRV_I2C_BUFFER_HANDLE bufferHandle, uintptr_t context)

DRV_I2C_BUFFER_HANDLE Type

File

drv_i2c.h

C

```
typedef uintptr_t DRV_I2C_BUFFER_HANDLE;
```

Description

This is type DRV_I2C_BUFFER_HANDLE.

DRV_I2C_BUS_LEVEL Enumeration

Lists the Operational Voltage level of I2C.

File

drv_i2c.h

C

```
typedef enum {
  DRV_I2C_OPEN_COLLECTOR_LEVEL,
  DRV_I2C_SMBus_LEVEL
} DRV_I2C_BUS_LEVEL;
```

Members

Members	Description
DRV_I2C_OPEN_COLLECTOR_LEVEL	I2C BUS LEVEL
DRV_I2C_SMBus_LEVEL	SMBus level

Description

I2C Bus Levels

This enumeration lists if the I2C is configured to operate in the traditional I2C mode or the SMBus mode. *

Remarks

None.

DRV_I2C_BUS_SPEED Enumeration

Lists the I2C Bus speed mode.

File

drv_i2c.h

C

```
typedef enum {
  DRV_I2C_NORMAL_SPEED,
  DRV_I2C_HIGH_SPEED
} DRV_I2C_BUS_SPEED;
```

Description

I2C Bus Speed

This enumeration lists if the I2C is configured to operate at High-Speed or Normal Speed.

Remarks

None.

DRV_I2C_CallBack Type

Points to a callback to initiate a particular function.

File

drv_i2c.h

C

```
typedef void (* DRV_I2C_CallBack)(DRV_I2C_BUFFER_EVENT event, void * context);
```

Description

I2C Buffer Event Callback

This type identifies the I2C Buffer Event. It allows the client driver to register a callback using DRV_I2C_BUFFER_EVENT. By using this mechanism, the driver client will can initiate an operation. This is intended to be used for SLAVE mode because the Master drives the I2C bus and through this the Slave can issue a READ or WRITE based on the status of R/W bit received from the Master

Remarks

A transfer can be composed of various transfer segments. Once a transfer is completed the driver will call the client registered transfer callback.

The callback could be called from ISR context and should be kept as short as possible. It is meant for signaling and it should not be blocking.

Parameters

Parameters	Description
DRV_I2C_BUFFER_EVENT	Status of I2C transfer
context	This is left for future implementation

Function

typedef void (*DRV_I2C_CallBack)(DRV_I2C_BUFFER_EVENT event, void * context)

DRV_I2C_INIT Structure

Identifies the initialization values that are passed as parameters to the initialize and reinitialize routines of the I2C module.

File

drv_i2c.h

C

```
typedef struct _DRV_I2C_INIT {
  SYS_MODULE_INIT moduleInit;
  I2C_MODULE_ID i2cId;
  DRV_I2C_MODE i2cMode;
  DRV_I2C_ADDRESS_WIDTH addWidth;
  bool reservedaddenable;
  bool generalcalladdress;
  I2C_SLAVE_ADDRESS_VALUE slaveaddvalue;
  uint32_t baudRate;
  DRV_I2C_BUS_LEVEL buslevel;
  DRV_I2C_BUS_SPEED busspeed;
  INT_SOURCE mstrInterruptSource;
  INT_SOURCE slaveInterruptSource;
  INT_SOURCE errInterruptSource;
  unsigned int queueSize;
  DRV_I2C_CallBack operationStarting;
  DRV_I2C_SLAVE_ADDRESS_MASK maskslaveaddress;
} DRV_I2C_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
I2C_MODULE_ID i2cId;	Identifies peripheral (PLIB-level) ID
DRV_I2C_MODE i2cMode;	I2C Usage Mode Type
DRV_I2C_ADDRESS_WIDTH addWidth;	Communication Width
bool reservedaddenable;	Reserved Address rule enable
bool generalcalladdress;	General Call address enable
I2C_SLAVE_ADDRESS_VALUE slaveaddvalue;	Slave address value
uint32_t baudRate;	Baud Rate Value
DRV_I2C_BUS_LEVEL buslevel;	I2C Clock mode
DRV_I2C_BUS_SPEED busspeed;	I2C Bus Speed
INT_SOURCE mstrInterruptSource;	Master Interrupt Source for I2C module
INT_SOURCE slaveInterruptSource;	Slave Interrupt Source for I2C module
INT_SOURCE errInterruptSource;	Error Interrupt Source for I2C module
unsigned int queueSize;	This is the buffer queue size. This is the maximum number of transfer requests that driver will queue. For a static build of the driver, this is overridden by the DRV_I2C_QUEUE_SIZE macro in system_config.h
DRV_I2C_CallBack operationStarting;	This callback is fired when an operation needs to be initiated on the I2C bus. This callback is mainly intended when the driver is used in SLAVE mode and is required to send data to the Master. The callback signals the application to prepare data for transfer to Master. This callback may be called from an ISR so should not include OSAL calls. The context parameter is the same one passed into the BufferAddRead, BufferAddWrite, BufferAddWriteRead function.
DRV_I2C_SLAVE_ADDRESS_MASK maskslaveaddress;	Slave address Mask value, the I2C slave address match ignored for the bit fields that the mask is set

Description

I2C Initialization Values

This enumeration identifies the values that are passed as parameters to the initialize and reinitialize routines of the I2C module.

These values specify different I2C module initialization parameters.

DRV_I2C_INIT_CODE Enumeration

Codes used for an I2C module initialization.

File

drv_i2c.h

C

```
typedef enum {
  DRV_I2C_INIT_DATA_NONE = 0,
  DRV_I2C_INIT_DATA_MASTER = 0x01,
  DRV_I2C_INIT_DATA_SLAVE = 0x02
} DRV_I2C_INIT_CODE;
```

Members

Members	Description
DRV_I2C_INIT_DATA_NONE = 0	No specific I2C initialization data present just system power state request
DRV_I2C_INIT_DATA_MASTER = 0x01	Initialize the I2C module as master, master data is to follow
DRV_I2C_INIT_DATA_SLAVE = 0x02	Initialize the I2C module as slave, slave data is to follow

Description

I2C Device Initialization Code

This enumeration identifies the codes that can be used to specify I2C module initialization. They are used in the system calls DRV_I2C_Initialize and DRV_I2C_Reinitialize as part of the specific module SYS_MODULE_INIT.moduleCode value.

The values act like flags and multiple flags can be set.

Remarks

The initialization code is I2C module-specific. It can be extended to carry additional information.

The system power code is always present and has to be processed. The device specific information can be missing if the system wants to change a power state only.

DRV_I2C_MODE Enumeration

Lists the operation mode of I2C module.

File

drv_i2c.h

C

```
typedef enum {
  DRV_I2C_MODE_MASTER,
  DRV_I2C_MODE_SLAVE
} DRV_I2C_MODE;
```

Members

Members	Description
DRV_I2C_MODE_MASTER	I2C Mode Master
DRV_I2C_MODE_SLAVE	I2C Mode Slave

Description

I2C Operation Mode

This enumeration lists if the I2C module is configured as a Master or a Slave.

Remarks

None.

DRV_I2C_SLAVE_ADDRESS_MASK Type

File

drv_i2c.h

C

```
typedef uint8_t DRV_I2C_SLAVE_ADDRESS_MASK;
```

Description

HV104_0217 Added variable for Slave address Mask

© 2013-2015 Microchip Technology Inc.

DRV_I2C_TRANSFER_HANDLE Type

Handle to an ongoing I2C transfer.

File

drv_i2c.h

C

```
typedef const void * DRV_I2C_TRANSFER_HANDLE;
```

Description

I2C Transfer Handle

This handle identifies an ongoing I2C transfer. The handle allows the driver client to check the status of that particular transfer by calling DRV_I2C_TransferStatus() or DRV_I2C_TransferStatusDcpt().

The status of an ongoing transfer can be monitored by using the transfer handle. Refer to DRV_I2C_TransferRegisterCallback function.

Remarks

This handle is provided by the routine that schedules a transfer. It is valid from the time a transfer has been enqueued until the time that the transfer has completed (and has been optionally acknowledged). At that time the handle becomes invalid.

Exception: If the transfer is marked persistent with DRV_I2C_TRANSFER_FLAG_PERSISTENT, the handle is valid across multiple calls, even after a transfer is completed. Even a persistent transfer can be removed from the driver's queues by using DRV_I2C_TransferRemove

I2C_DATA_TYPE Type

File

drv_i2c.h

C

```
typedef unsigned char I2C_DATA_TYPE;
```

Description

This is type I2C_DATA_TYPE.

I2C_SLAVE_ADDRESS_7bit Type

File

drv_i2c.h

C

```
typedef uint8_t I2C_SLAVE_ADDRESS_7bit;
```

Description

need to type cast 10-bit slave address into 8 bit mode

I2C_SLAVE_ADDRESS_VALUE Type

File

drv_i2c.h

C

```
typedef uint16_t I2C_SLAVE_ADDRESS_VALUE;
```

Description

This is type I2C_SLAVE_ADDRESS_VALUE.

DRV_I2C_INDEX_0 Macro

The number of I2C instances that are present on the device. This number is dependent on the type of device.

File

drv_i2c.h

C

```
#define DRV_I2C_INDEX_0 0
```

Description

DRV_I2C_INDEX

DRV_I2C_INDEX_1 Macro

File

drv_i2c.h

C

```
#define DRV_I2C_INDEX_1 1
```

Description

This is macro DRV_I2C_INDEX_1.

DRV_I2C_INDEX_2 Macro

File

drv_i2c.h

C

```
#define DRV_I2C_INDEX_2 2
```

Description

This is macro DRV_I2C_INDEX_2.

DRV_I2C_INDEX_3 Macro

File

drv_i2c.h

C

```
#define DRV_I2C_INDEX_3 3
```

Description

This is macro DRV_I2C_INDEX_3.

DRV_I2C_INDEX_4 Macro

File

drv_i2c.h

C

```
#define DRV_I2C_INDEX_4 4
```

Description

This is macro DRV_I2C_INDEX_4.

DRV_I2C_INDEX_5 Macro

File

drv_i2c.h

C

```
#define DRV_I2C_INDEX_5 5
```

Description

This is macro DRV_I2C_INDEX_5.

DRV_I2C_BUFFER_QUEUE_SUPPORT Macro

Specifies if the Buffer Queue support should be enabled.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_BUFFER_QUEUE_SUPPORT false
```

Description

I2C Driver Buffer Queue Support

This macro defines if Buffer Queue support should be enabled. Setting this macro to true will enable buffer queue support and all buffer related driver function.

Remarks

None

DRV_I2C_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_INSTANCES_NUMBER 5
```

Description

I2C driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of I2C modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None

DRV_I2C_INTERRUPT_MODE Macro

Macro controls interrupt based operation of the driver

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_INTERRUPT_MODE true
```

Description

I2C Interrupt Mode Operation Control

This macro controls the interrupt based operation of the driver. The possible values it can take are

- true - Enables the interrupt mode
- false - Enables the polling mode

If the macro value is true, then Interrupt Service Routine for the interrupt should be defined in the application. The DRV_I2C_Tasks() routine should be called in the ISR.

Remarks

None

DRV_I2C_QUEUE_DEPTH_COMBINED Macro

Number of entries of all queues in all instances of the driver.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_QUEUE_DEPTH_COMBINED 10
```

Description

I2C Driver Instance combined queue depth.

This macro defines the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for transmit and receive operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build). The hardware instance transmit buffer queue will queue transmit buffers submitted by the DRV_I2C_BufferAddWrite() function. The hardware instance receive buffer queue will queue receive buffers submitted by the DRV_I2C_BufferAddRead() function.

A buffer queue will contain buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all I2C driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking read and write requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its transmit and receive buffer queue size.

As an example, consider the case of static single client driver application where full duplex non blocking operation is desired without queuing, the minimum transmit queue depth and minimum receive queue depth should be 1. Hence the total number of buffer entries should be 2.

In the current implementation of I2C driver, queueing of Buffers is not supported. This will be added in a future release.

Remarks

None

Files

Files

Name	Description
drv_i2c.h	I2C module driver interface header.
drv_i2c_config_template.h	I2C device driver configuration file.

Description

drv_i2c.h

I2C module driver interface header.

Enumerations

	Name	Description
	DRV_I2C_ADDRESS_WIDTH	Lists the Address Width of the Slave.
	DRV_I2C_BUFFER_EVENT	Lists the different conditions that happens during a buffer transfer.
	DRV_I2C_BUS_LEVEL	Lists the Operational Voltage level of I2C.
	DRV_I2C_BUS_SPEED	Lists the I2C Bus speed mode.
	DRV_I2C_INIT_CODE	Codes used for an I2C module initialization.
	DRV_I2C_MODE	Lists the operation mode of I2C module.

Functions

	Name	Description
	DRV_I2C_BaudRateSet	Sets the Baud-Rate for the I2C module. **Implementation:** Static
	DRV_I2C_BufferAddRead	This function reads data written from either Master or Slave. **Implementation:** Dynamic
	DRV_I2C_BufferAddWrite	This function writes data to Master or Slave. **Implementation:** Dynamic
	DRV_I2C_BufferAddWriteRead	This function writes data to Slave, inserts restart and requests read from slave. **Implementation:** Dynamic
	DRV_I2C_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_I2C_BufferStatus	Returns status of data transfer when Master or Slave acts either as a transmitter or a receiver. **Implementation:** Dynamic
	DRV_I2C_ByteRead	Reads data from the receive register. **Implementation:** Static
	DRV_I2C_ByteWrite	Writes data byte to the appropriate module. **Implementation:** Static
	DRV_I2C_Close	Closes an opened instance of an I2C module driver. **Implementation:** Dynamic
	DRV_I2C_Deinitialize	Deinitializes the index instance of the I2C module. **Implementation:** Static/Dynamic

	Name	Description
	DRV_I2C_Initialize	Initializes hardware and data for the index instance of the I2C module. **Implementation:** Static/Dynamic
	DRV_I2C_IRQEventSend	IRQ line logic to let Master know that Slave has data to be sent. **Implementation:** Dynamic
	DRV_I2C_MasterACKSend	Sends an ACK to the slave. **Implementation:** Static
	DRV_I2C_MasterBusIdle	Checks if the I2C bus is idle. **Implementation:** Static
	DRV_I2C_MasterNACKSend	Sends an ACK to the slave. **Implementation:** Static
	DRV_I2C_MasterRestart	Issues a START on I2C bus. **Implementation:** Static
	DRV_I2C_MasterStart	Issues a START on I2C bus. **Implementation:** Static
	DRV_I2C_MasterStop	Issues a STOP on I2C bus. **Implementation:** Static
	DRV_I2C_Open	Opens the specified instance of the I2C driver for use and provides an "open-instance" handle. **Implementation:** Dynamic
	DRV_I2C_ReceiverBufferIsEmpty	Checks if no byte is present in Receive register. **Implementation:** Static
	DRV_I2C_RestartEventSend	Master sends STOP condition on I2C bus indicating end of WRITE or READ. **Implementation:** Dynamic
	DRV_I2C_SetUpByteRead	Prepare Master or Slave for receiving data. **Implementation:** Static
	DRV_I2C_Status	Provides the current status of the index instance of the I2C module. **Implementation:** Dynamic
	DRV_I2C_StopEventSend	Master sends STOP condition on I2C bus indicating end of WRITE or READ. **Implementation:** Dynamic
	DRV_I2C_Tasks	Maintains the State Machine of the I2C driver and performs all the protocol level actions. **Implementation:** Dynamic
	DRV_I2C_WaitForACKOrNACKComplete	Waits for ACK/NACK sequence to be complete. **Implementation:** Static
	DRV_I2C_WaitForByteWriteToComplete	Checks for the write operation to be completed. **Implementation:** Static
	DRV_I2C_WaitForReadByteAvailable	Checks if data is available in the receive register. **Implementation:** Static
	DRV_I2C_WaitForStartComplete	Checks if a successful START was issued on the I2C bus. **Implementation:** Static
	DRV_I2C_WaitForStopComplete	Checks if a successful STOP was issued on the I2C bus. **Implementation:** Static
	DRV_I2C_WriteByteAcknowledged	Checks if byte written was ACK'ed or NACK'ed. **Implementation:** Static

Macros

	Name	Description
	DRV_I2C_INDEX_0	The number of I2C instances that are present on the device. This number is dependent on the type of device.
	DRV_I2C_INDEX_1	This is macro DRV_I2C_INDEX_1.
	DRV_I2C_INDEX_2	This is macro DRV_I2C_INDEX_2.
	DRV_I2C_INDEX_3	This is macro DRV_I2C_INDEX_3.
	DRV_I2C_INDEX_4	This is macro DRV_I2C_INDEX_4.
	DRV_I2C_INDEX_5	This is macro DRV_I2C_INDEX_5.

Structures

	Name	Description
	_DRV_I2C_INIT	Identifies the initialization values that are passed as parameters to the initialize and reinitialize routines of the I2C module.
	DRV_I2C_INIT	Identifies the initialization values that are passed as parameters to the initialize and reinitialize routines of the I2C module.

Types

	Name	Description
	DRV_I2C_BUFFER_EVENT_HANDLER	Points to a callback after completion of an I2C transfer.
	DRV_I2C_BUFFER_HANDLE	This is type DRV_I2C_BUFFER_HANDLE.
	DRV_I2C_CallBack	Points to a callback to initiate a particular function.
	DRV_I2C_SLAVE_ADDRESS_MASK	HV104_0217 Added variable for Slave address Mask
	DRV_I2C_TRANSFER_HANDLE	Handle to an ongoing I2C transfer.
	I2C_DATA_TYPE	This is type I2C_DATA_TYPE.
	I2C_SLAVE_ADDRESS_7bit	need to type cast 10-bit slave address into 8 bit mode
	I2C_SLAVE_ADDRESS_VALUE	This is type I2C_SLAVE_ADDRESS_VALUE.

Description

I2C Device Driver Interface Header File

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the I2C module driver.

File Name

drv_i2c.h

Company

Microchip Technology Inc.

drv_i2c_config_template.h

I2C device driver configuration file.

Macros

	Name	Description
	DRV_DYNAMIC_BUILD	Dynamic driver build, dynamic device instance parameters.

	DRV_I2C_BUFFER_QUEUE_SUPPORT	Specifies if the Buffer Queue support should be enabled.
	DRV_I2C_CONFIG_BUILD_TYPE	Selects static or dynamic driver build configuration.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BASIC	Enables the device driver to support basic transfer mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BLOCKING	Enables the device driver to support blocking operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_EXCLUSIVE	Enables the device driver to support operation in Exclusive mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_MASTER	Enables the device driver to support operation in Master mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_NON_BLOCKING	Enables the device driver to support non-blocking during operations
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_READ	Enables the device driver to support read operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_SLAVE	Enables the device driver to support operation in Slave mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE	Enables the device driver to support write operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE_READ	Enables the device driver to support write followed by read.
	DRV_I2C_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_I2C_INTERRUPT_MODE	Macro controls interrupt based operation of the driver
	DRV_I2C_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.
	DRV_STATIC_BUILD	Static driver build, static device instance parameters.

Description

I2C Device Driver Configuration

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_i2c_config.h

Company

Microchip Technology Inc.

I2S Driver Library Help

This topic describes the I2S Driver Library.

Introduction

This library provides an interface to manage the Audio Protocol Interface Modes of the Serial Peripheral Interface (SPI) module on the Microchip family of microcontrollers.

Description

The SPI module can be interfaced to most available codec devices to provide microcontroller-based audio solutions. The SPI module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- SDIx: Serial Data Input for receiving sample digital audio data (ADCDAT)
- SDOx: Serial Data Output for transmitting digital audio data (DACDAT)
- SCKx: Serial Clock, also known as bit clock (BCLK)
- /SSx: Left/Right Channel Clock (LRCK)

BCLK provides the clock required to drive the data out or into the module, while LRCK provides the synchronization of the frame based on the protocol mode selected.

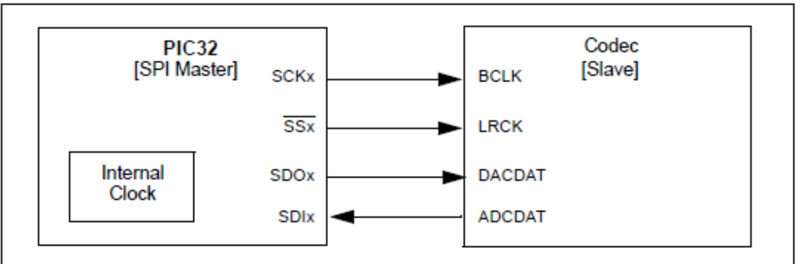
In Master mode, the module generates both the BCLK on the SCKx pin and the LRCK on the /SSx pin. In certain devices, while in Slave mode, the module receives these two clocks from its I2S partner, which is operating in Master mode.

When configured in Master mode, the leading edge of SCK and the LRCK are driven out within one SCK period of starting the audio protocol. Serial data is shifted in or out with timings determined by the protocol mode set.

In Slave mode, the peripheral drives zeros out SDO, but does not transmit the contents of the transmit FIFO until it sees the leading edge of the LRCK, after which time it starts receiving data.

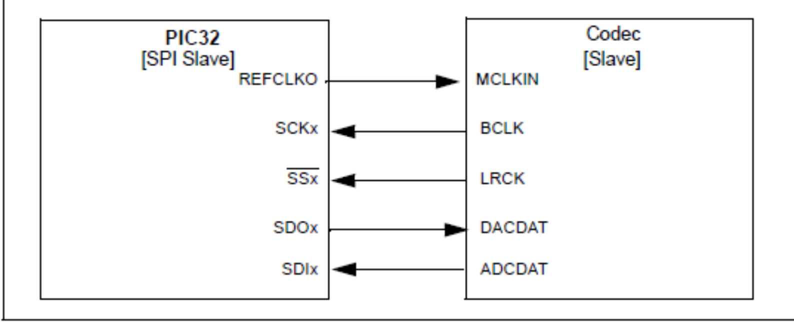
Master Mode

Master Generating its Own Clock – Output BCLK and LRCK



Slave Mode

Codec Device as Master Derives MCLK from PIC32 Reference Clock Out



Audio Protocol Modes

The SPI module supports four audio protocol modes and can be operated in any one of these modes:

- I2S mode

- Left-Justified mode
- Right-Justified mode
- PCM/DSP mode

Using the Library

This topic describes the basic architecture of the I2S Driver Library and provides information and examples on its use.

Description

Interface Header File: drv_i2s.h

The interface to the I2S Driver Library is defined in the drv_i2s.h header file. Any C language source (.c) file that uses the I2S Driver Library should include drv_i2s.h.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

The SPI Peripheral Library provides the low-level abstraction of the SPI module on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the I2S Driver Library interface.

Description

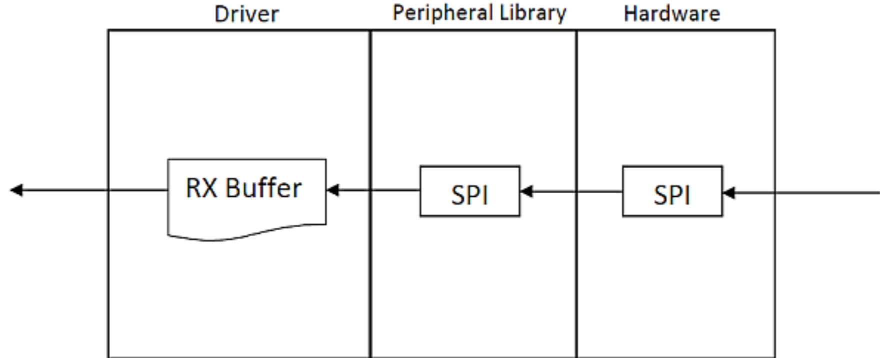
I2S Software Abstraction Block Diagram

Different types of SPIs are available on Microchip microcontrollers. Some have an internal buffer mechanism and some do not. The buffer depth varies across part families. The SPI Peripheral Library provides the ability to access these buffers. The I2S Driver Library abstracts out these differences and provides a unified model for audio data transfer across different types of SPI modules.

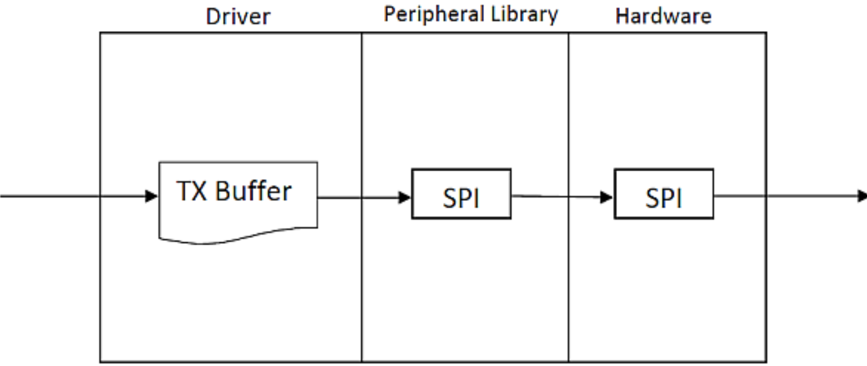
Both the transmitter and receiver provide a buffer in the driver, which transmits and receives data to/from the hardware. The I2S Driver Library provides a set of interfaces to perform the read and the write.

The following diagrams illustrate the model used by the I2S Driver Library for the transmitter and receiver.

Receiver Abstraction Model



Transmitter Abstraction Model



© 2013-2015 Microchip Technology Inc.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The I2S driver library provides an API interface to transfer/receive digital audio data using supported Audio protocols. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the I2S Driver Library.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open and close functions.
Data Transfer Functions	Provides data transfer functions.
Miscellaneous Functions	Provides driver miscellaneous functions such as baud rate setting, get error functions, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the I2S Driver Library.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

Note: Not all modes are available on all devices. Please refer to the specific device data sheet to determine the supported modes.

© 2013-2015 Microchip Technology Inc.

System Access

This section provides information on system access.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the I2S module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_I2S_INIT or by using Initialization Overrides) that are supported by the specific I2S device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- The actual peripheral ID enumerated as the PLIB level module ID (e.g., SPI_ID_2)
- Defining the respective interrupt sources for TX, RX, DMA TX Channel, DMA RX Channel and Error Interrupt

The DRV_I2S_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_I2S_Deinitialize, DRV_I2S_Status, DRV_I2S_Tasks, and DRV_I2S_TasksError.

Notes:

1. The system initialization setting only effect the instance of the peripheral that is being initialized.
2. Configuration of the dynamic driver for DMA mode(uses DMA channel for data transfer) or Non DMA mode can be performed by appropriately setting the 'dmaChannelTransmit' and 'dmaChannelReceive' variables of the DRV_I2S_INIT structure. For example the TX will be in DMA mode when 'dmaChannelTransmit' is initialized to a valid supported channel number from the enum DMA_CHANNEL. TX will be in Non DMA mode when 'dmaChannelTransmit' is initialized to 'DMA_CHANNEL_NONE'.

Example:

```
    DRV_I2S_INIT                  init;
    SYS_MODULE_OBJ              objectHandle;

    init.moduleInit.value           = SYS_MODULE_POWER_RUN_FULL;
    init.spiID                      = SPI_ID_1;
    init.usageMode                  = DRV_I2S_MODE_MASTER;
    init.baudClock                  = SPI_BAUD_RATE_MCLK_CLOCK;
    init.baud                       = 48000;
    init.clockMode                  = DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_FALL;
    init.audioCommWidth             = SPI_AUDIO_COMMUNICATION_24DATA_32FIFO_32CHANNEL;
    init.audioTransmitMode          = SPI_AUDIO_TRANSMIT_STEREO;
    init.inputSamplePhase           = SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE;
    init.protocolMode               = DRV_I2S_AUDIO_I2S;
    init.txInterruptSource          = INT_SOURCE_SPI_1_TRANSMIT;
    init.rxInterruptSource          = INT_SOURCE_SPI_1_RECEIVE;
    init.errorInterruptSource       = INT_SOURCE_SPI_1_ERROR;
    init.queueSizeTransmit          = 3;
    init.queueSizeReceive           = 2;
    init.dmaChannelTransmit         = DMA_CHANNEL_NONE;
    init.dmaChannelReceive          = DMA_CHANNEL_NONE;

  objectHandle = DRV_I2S_Initialize(DRV_I2S_INDEX_1, (SYS_MODULE_INIT*)init);
    if (SYS_MODULE_OBJ_INVALID == objectHandle)
    {
        // Handle error
    }
```

Task Routine

In a polled environment, the system will call DRV_I2S_Tasks and DRV_I2S_TasksError from the System Task Service. In an interrupt-based implementation, DRV_I2S_Tasks and DRV_I2S_TasksError will be called from the Interrupt Service Routine of the I2S. When a DMA channel is used for transmission/reception DRV_I2S_Tasks and DRV_I2S_TasksError will be internally called by the driver from the DMA channel event handler.

Client Access

This section provides information on general client operation.

Description

General Client Operation

For the application to start using an instance of the module, it must call the DRV_I2S_Open function. This provides the settings required to open the I2S instance for operation. If the driver is deinitialized using the function DRV_I2S_Deinitialize, the application must call the DRV_I2S_Open function again to set up the instance of the I2S.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Example:

```
DRV_HANDLE handle;
handle = DRV_I2S_Open(DRV_I2S_INDEX_0, (DRV_IO_INTENT_WRITE | DRV_IO_INTENT_NONBLOCKING));
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Client Operations - Buffered

This section provides information on buffered client operations.

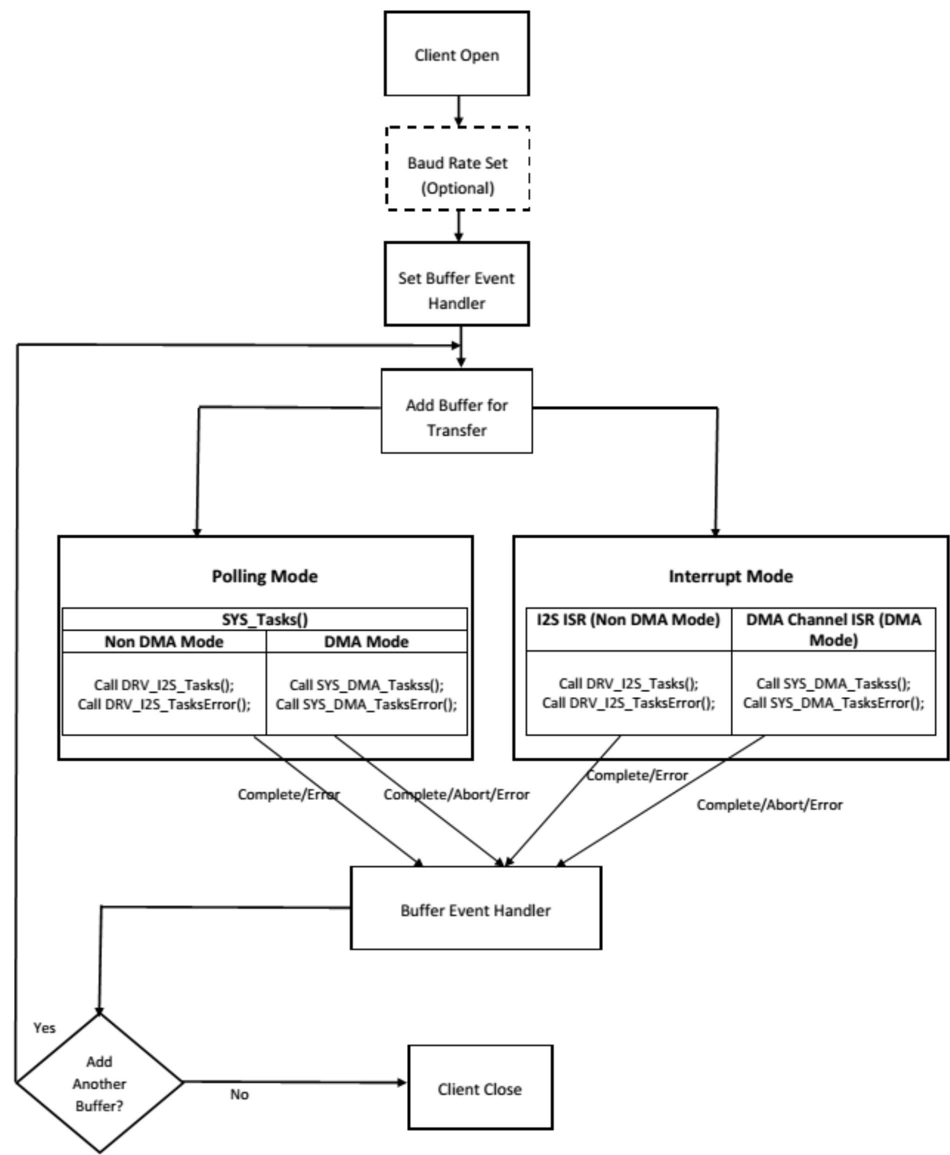
Description

Client Operations - Buffered

Client buffered operations provide a the typical audio interface. The functions DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite, and DRV_I2S_BufferAddWriteRead are the buffered data operation functions. The buffered functions schedules non-blocking operations. The function adds the request to the hardware instance queues and returns a buffer handle. The requesting client also registers a callback event with the driver. The driver notifies the client with DRV_I2S_BUFFER_EVENT_COMPLETE, DRV_I2S_BUFFER_EVENT_ERROR or DRV_I2S_BUFFER_EVENT_ABORT events.

The buffer add requests are processed under DRV_I2S_Tasks, DRV_I2S_TasksError functions. These functions are called from the I2S channel ISR in interrupt mode or from SYS_Tasks routine in Polled mode. When a DMA channel is used for transmission/reception DRV_I2S_Tasks and DRV_I2S_TasksError will be internally called by the driver from the DMA channel event handler.

The following diagram illustrates the buffered data operations



Note: It is not necessary to close and reopen the client between multiple transfers.

An application using the buffered functionality needs to perform the following steps:

1. The system should have completed necessary setup and initializations.
2. If DMA mode is desired, the DMA should be initialized by calling SYS_DMA_Initialize.
3. The necessary ports setup and remapping must be done for I2S lines: ADCDAT, DACDAT, BCLK, LRCK and MCLK (if required).
4. The driver object should have been initialized by calling DRV_I2S_Initialize. If DMA mode is desired, related attributes in the init structure must be set.
5. Open the driver using DRV_I2S_Open with the necessary ioIntent to get a client handle.
6. The necessary BCLK, LRCK, and MCLK should be set up so as to generate the required media bit rate.
7. The necessary Baud rate value should be set up by calling DRV_I2S_BaudrateSet.
8. The Register and event handler for the client handle should be set up by calling DRV_I2S_BufferEventHandlerSet.
9. Add a buffer to initiate the data transfer by calling DRV_I2S_BufferAddWrite/DRV_I2S_BufferAddRead/DRV_I2S_BufferAddWriteRead.

10. Based on polling or interrupt mode service the data processing should be set up by calling DRV_I2S_Tasks, DRV_I2S_TasksError from system tasks or I^2S ISR. When a DMA channel is used for transmission/reception system calls SYS_DMA_Tasks(), SYS_DMA_TasksError() from the system tasks or DMA channel ISR, DRV_I2S_Tasks and DRV_I2S_TasksError will be internally called by the driver from the DMA channel event handler.
11. Repeat step 9 through step 10 to handle multiple buffer transmission and reception.
12. When the client is done it can use DRV_I2S_Close to close the client handle.

Example 1:

```
// The following is an example for a Polled mode buffered transmit

#define SYS_I2S_DRIVER_INDEX DRV_I2S_1 // I2S Uses SPI Hardware
#define BUFFER_SIZE 1000
// I2S initialization structure.
// This should be populated with necessary settings.
// attributes dmaChannelTransmit/dmaChannelReceive
// and dmaInterruptTransmitSource/dmaInterruptReceiveSource
// must be set if DMA mode of operation is desired.
DRV_I2S_INIT i2sInit;
SYS_MODULE_OBJ sysObj;      //I2S module object
DRV_HANDLE handle;          //Client handle
uint32_t i2sClock;          //BCLK frequency
uint32_t baudrate;          //baudrate
uint16_t myAudioBuffer[BUFFER_SIZE]; //Audio buffer to be transmitted
DRV_I2S_BUFFER_HANDLE bufferHandle;
APP_DATA_S state;           //Application specific state
uintptr_t contextHandle;

void SYS_Initialize ( void* data )
{
    // The system should have completed necessary setup and initializations.
    // Necessary ports setup and remapping must be done for I2S lines ADCDAT,
    // DACDAT, BCLK, LRCK and MCLK

    sysObj = DRV_I2S_Initialize(SYS_I2S_DRIVER_INDEX, (SYS_MODULE_INIT*)&i2sInit);
    if (SYS_MODULE_OBJ_INVALID == sysObj)
    {
        // Handle error
    }
}


void App_Task(void)
{
        switch(state)
        {
            case APP_STATE_INIT:
            {
                handle = DRV_I2S_Open(SYS_I2S_DRIVER_INDEX, (DRV_IO_INTENT_WRITE |
DRV_IO_INTENT_NONBLOCKING));
                if(handle != DRV_HANDLE_INVALID )
                {
                    /* Update the state */
                    state = APP_STATE_WAIT_FOR_READY;
                }
            }
            break;

            case  APP_STATE_WAIT_FOR_READY:
            {
                // Necessary clock settings must be done to generate
                // required MCLK, BCLK and LRCK
                DRV_I2S_BaudrateSet(handle, i2sClock, baudrate);

                /* Set the Event handler */
                DRV_I2S_BufferEventHandlerSet(handle,App_BufferEventHandler,
                contextHandle);
```

```
                /* Add a buffer to write*/
                DRV_I2S_BufferAddWrite(handle, &bufferHandle
                                        myAudioBuffer, BUFFER_SIZE);
                if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
                {
                    // Error handling here
                }
                state = APP_STATE_IDLE;
            }
            break;

            case APP_STATE_WAIT_FOR_DONE:
                state = APP_STATE_DONE;
            break;

            case  APP_STATE_DONE:
                // Close done
                DRV_I2S_Close(handle);
            break;

            case  APP_STATE_IDLE:
                // Do nothing
            break;

            default:
            break;
        }
}


void App_BufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    uint8_t temp;

    if(DRV_I2S_BUFFER_EVENT_COMPLETE == event)
    {
        // Can set state = APP_STATE_WAIT_FOR_DONE;
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ERROR == event)
    {
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ABORT == event)
    {
        // Take Action as needed
    }
    else
    {
        // Do nothing
    }
}


void SYS_Tasks ( void )
{
    DRV_I2S_Tasks((SYS_MODULE_OBJ)sysObj);
    DRV_I2S_TasksError((SYS_MODULE_OBJ)sysObj);

    /* Call the application's tasks routine */
    APP_Tasks ( );
}
```

Example 2:

```
// The following is an example for interrupt mode buffered transmit
```

```
#define SYS_I2S_DRIVER_INDEX DRV_I2S_1 // I2S Uses SPI Hardware
#define BUFFER_SIZE 1000
// I2S initialization structure.
// This should be populated with necessary settings.
// attributes dmaChannelTransmit/dmaChannelReceive
// and dmaInterruptTransmitSource/dmaInterruptReceiveSource
// must be set if DMA mode of operation is desired.
DRV_I2S_INIT i2sInit;
SYS_MODULE_OBJ sysObj; //I2S module object
DRV_HANDLE handle;  //Client handle
uint32_t i2sClock;   //BCLK frequency
uint32_t baudrate;   //baudrate
uint16_t myAudioBuffer[BUFFER_SIZE]; //Audio buffer to be transmitted
DRV_I2S_BUFFER_HANDLE bufferHandle;
APP_DATA_S state; //Application specific state
uintptr_t contextHandle;

void SYS_Initialize ( void* data )
{
    // The system should have completed necessary setup and initializations.
    // Necessary ports setup and remapping must be done for I2S lines ADCDAT,
    // DACDAT, BCLK, LRCK and MCLK

    sysObj = DRV_I2S_Initialize(SYS_I2S_DRIVER_INDEX, (SYS_MODULE_INIT*)&i2sInit);
    if (SYS_MODULE_OBJ_INVALID == sysObj)
    {
        // Handle error
    }
}


void App_Task(void)
{
        switch(state)
        {
            case APP_STATE_INIT:
            {
                handle = DRV_I2S_Open(SYS_I2S_DRIVER_INDEX, (DRV_IO_INTENT_WRITE |
DRV_IO_INTENT_NONBLOCKING));
                if(handle != DRV_HANDLE_INVALID )
                {
                    /* Update the state */
                    state = APP_STATE_WAIT_FOR_READY;
                }
            }
            break;

            case  APP_STATE_WAIT_FOR_READY:
            {
                // Necessary clock settings must be done to generate
                // required MCLK, BCLK and LRCK
                DRV_I2S_BaudrateSet(handle, i2sClock, baudrate);

                /* Set the Event handler */
                DRV_I2S_BufferEventHandlerSet(handle,App_BufferEventHandler,
                contextHandle);

                /* Add a buffer to write*/
                DRV_I2S_BufferAddWrite(handle, &bufferHandle
                                        myAudioBuffer, BUFFER_SIZE);
                if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
                {
                    // Error handling here
                }
                state = APP_STATE_IDLE;
            }
            break;

            case APP_STATE_WAIT_FOR_DONE:
```

```
                state = APP_STATE_DONE;
            break;


            case  APP_STATE_DONE:
            {
                // Close done
                DRV_I2S_Close(handle);
            }
            break;

            case  APP_STATE_IDLE:
                // Do nothing
            break;

            default:
            break;
        }
}


void App_BufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    uint8_t temp;

    if(DRV_I2S_BUFFER_EVENT_COMPLETE == event)
    {
        // Can set state = APP_STATE_WAIT_FOR_DONE;
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ERROR == event)
    {
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ABORT == event)
    {
        // Take Action as needed
    }
    else
    {
        // Do nothing
    }
}


void SYS_Tasks ( void )
{
    /* Call the application's tasks routine */
    APP_Tasks ( );
}


void __ISR ( _SPI1_VECTOR ) _InterruptHandler_I2S1 ( void )
{
    // Call the "tasks" functions for I2S module
    DRV_I2S_Tasks((SYS_MODULE_OBJ)sysObj);
    DRV_I2S_TasksError((SYS_MODULE_OBJ)sysObj);
}


// If DMA Channel 1 was setup during initialization instead of the previous I2S ISR, the following should
be implemented
void __ISR ( _DMA1_VECTOR ) _InterruptHandler_DMA_CHANNEL_1 ( void )
{
    // Call the DMA system tasks which internally will call the I2S Tasks.
    SYS_DMA_Tasks((SYS_MODULE_OBJ)sysObj);
    SYS_DMA_TasksError((SYS_MODULE_OBJ)sysObj);
```

```
}
```

Client Operations - Non-buffered

This section provides information on non-buffered client operations.

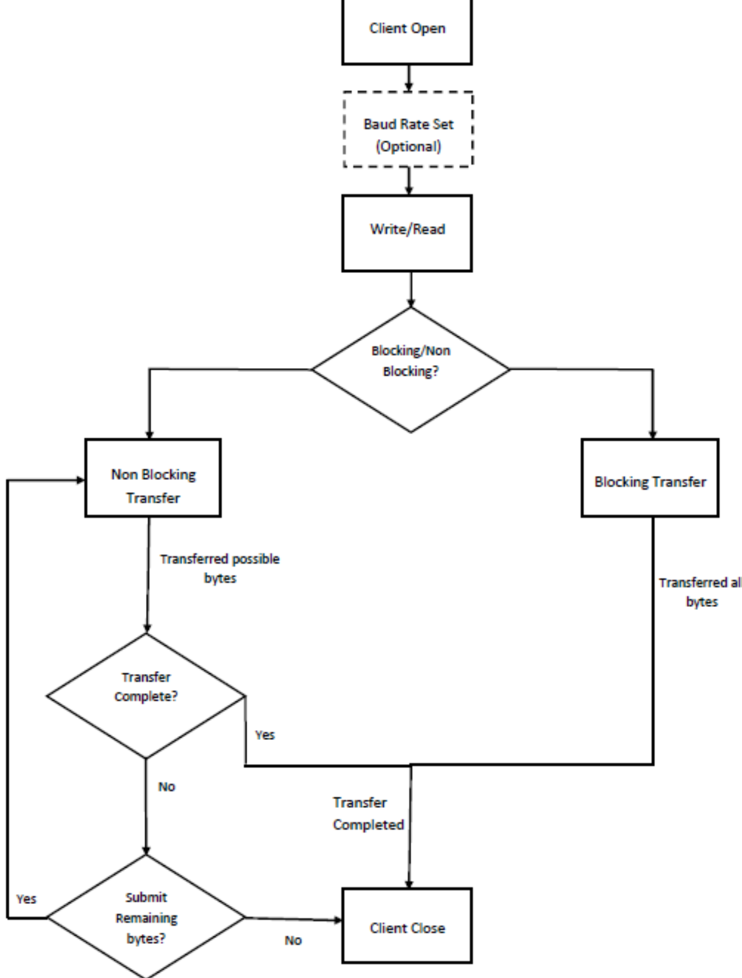
Description

Client Operations - Non-buffered

Client non-buffered operations provide a basic interface for the driver operation. This interface could be used by applications which have do not have buffered data transfer requirements. The functions DRV_I2S_Read and DRV_I2S_Write are the non-buffered data operation functions. The non-buffered functions are blocking/non-blocking depending upon the mode (ioIntent) the client was opened. If the client was opened for blocking mode these functions will only return when (or will block until) the specified data operation is completed or if an error occurred. If the client was opened for non-blocking mode, these functions will return with the number of bytes that were actually accepted for operation. The function will not wait until the data operation has completed.

Note: Non-buffered functions do not support interrupt/DMA mode.

The following diagram illustrates the non-buffered data operations



Note: It is not necessary to close and reopen the client between multiple transfers.

An application using the non-buffered functionality needs to perform the following steps:

1. The system should have completed necessary setup and initializations.
2. The necessary ports setup and remapping must be done for I2S lines: ADCDAT, DACDAT, BCLK, LRCK and MCLK (if required).

3. The driver object should have been initialized by calling DRV_I2S_Initialize.
4. Open the driver using DRV_I2S_Open with the necessary ioIntent to get a client handle.
5. The necessary BCLK, LRCK, and MCLK should be set up so as to generate the required media bit rate.
6. The necessary Baud rate value should be set up by calling DRV_I2S_BaudrateSet.
7. The Transmit/Receive data should be set up by calling DRV_I2S_Write/DRV_I2S_Read.
8. Repeat step 5 through step 7 to handle multiple buffer transmission and reception.
9. When the client is done it can use DRV_I2S_Close to close the client handle.

Example 1:

```
// The following is an example for a blocking transmit
    #define SYS_I2S_DRIVER_INDEX DRV_I2S_1 // I2S Uses SPI Hardware
    #define BUFFER_SIZE 1000
    DRV_I2S_INIT i2sInit; //I2S initialization structure
                          //This should be populated with necessary settings
    SYS_MODULE_OBJ sysObj; //I2S module object
    APP_DATA_S state; //Application specific state
    DRV_HANDLE handle;  //Client handle
    uint32_t i2sClock;   //BCLK frequency
    uint32_t baudrate;   //baudrate
    uint16_t myAudioBuffer[BUFFER_SIZE]; //Audio buffer to be transmitted
    uint32_t count;

    // The system should have completed necessary setup and initializations.
    // Necessary ports setup and remapping must be done for
    // I2S lines ADCDAT, DACDAT, BCLK, LRCK and MCLK

    sysObj = DRV_I2S_Initialize(SYS_I2S_DRIVER_INDEX, (SYS_MODULE_INIT*)&i2sInit);
    if (SYS_MODULE_OBJ_INVALID == sysObj)
    {
        // Handle error
    }
    while(1)
    {
        switch(state)
        {
            case APP_STATE_INIT:
            {
                handle = DRV_I2S_Open(SYS_I2S_DRIVER_INDEX, (DRV_IO_INTENT_WRITE | DRV_IO_INTENT_BLOCKING));
                if(handle != DRV_HANDLE_INVALID )
                {
                    /* Update the state */
                    state = APP_STATE_WAIT_FOR_READY;
                }
            }
            break;
            case  APP_STATE_WAIT_FOR_READY:
            {
                // Necessary clock settings must be done to generate
                // required MCLK, BCLK and LRCK
                DRV_I2S_BaudrateSet(handle, i2sClock, baudrate);
                // Blocks here and transfer the buffer
                count = DRV_I2S_Write(handle, &myAudioBuffer,BUFFER_SIZE);
                if(count == DRV_I2S_WRITE_ERROR)
                {
                    //Handle Error
                } else
                {
                    // Transfer Done
                    state = APP_STATE_DONE;
                }
            }
            break;
            case  APP_STATE_DONE:
            {
                // Close done
                DRV_I2S_Close(handle);
```

```
            }
            break;
            default:
            break;
        }
    }
```

Example 2:

```
// Following is an example for a non blocking transmit
    #define SYS_I2S_DRIVER_INDEX DRV_I2S_1 //I2S Uses SPI Hardware
    #define BUFFER_SIZE 1000
    DRV_I2S_INIT i2sInit; //I2S initialization structure.
                          // This should be populated with necessary settings
    SYS_MODULE_OBJ sysObj; //I2S module object
    APP_DATA_S state; //Application specific state
    DRV_HANDLE handle;  //Client handle
    uint32_t i2sClock;   //BCLK frequency
    uint32_t baudrate;   //baudrate
    uint16_t myAudioBuffer[BUFFER_SIZE]; //Audio buffer to be transmitted
    uint32_t count,total,size;

    total = 0;
    size = BUFFER_SIZE;

    // The system should have completed necessary setup and initializations.
    // Necessary ports setup and remapping must be done for I2S lines ADCDAT,
    // DACDAT, BCLK, LRCK and MCLK

    sysObj = DRV_I2S_Initialize(SYS_I2S_DRIVER_INDEX, (SYS_MODULE_INIT*)&i2sInit);
    if (SYS_MODULE_OBJ_INVALID == sysObj)
    {
        // Handle error
    }

    while(1)
    {
        switch(state)
        {
            case APP_STATE_INIT:
            {
                handle = DRV_I2S_Open(SYS_I2S_DRIVER_INDEX, (DRV_IO_INTENT_WRITE |
DRV_IO_INTENT_NONBLOCKING));
                if(handle != DRV_HANDLE_INVALID )
                {
                    /* Update the state */
                    state = APP_STATE_WAIT_FOR_READY;
                }
            }
            break;
            case  APP_STATE_WAIT_FOR_READY:
            {
                // Necessary clock settings must be done to generate
                // required MCLK, BCLK and LRCK
                DRV_I2S_BaudrateSet(handle, i2sClock, baudrate);
                // Transfer whatever possible number of bytes
                count = DRV_I2S_Write(handle, &myAudioBuffer,size);
                if(count == DRV_I2S_WRITE_ERROR)
                {
                    //Handle Error
                } else
                {
                    // 'count' bytes transferred
                    state = APP_STATE_WAIT_FOR_DONE;
                }
            }
            break;
            case APP_STATE_WAIT_FOR_DONE:
                // Can perform other Application tasks here
```

```
                // .........................
                // .........................
                // .........................
                size = size - count;
                if(size!=0)
                {
                    // Change the state so as to submit
                    // another possible transmission
                    state = APP_STATE_WAIT_FOR_READY;
                }
                else
                {
                    // We are done
                    state = APP_STATE_DONE;
                }
            break;
            case  APP_STATE_DONE:
            {
                if(DRV_I2S_CLOSE_FAILURE == DRV_I2S_Close(handle))
                {
                    // Handle error
                }
                else
                {
                    // Close done
                }
            }
            break;
            default:
            break;
        }
    }
```

Configuring the Library

Client Configuration

	Name	Description
	DRV_I2S_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_I2S_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.

System Configuration

	Name	Description
	DRV_I2S_INDEX	I2S Static Index selection
	DRV_I2S_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_I2S_INTERRUPT_MODE	Macro controls interrupt based operation of the driver
	DRV_I2S_INTERRUPT_SOURCE_ERROR	Defines the interrupt source for the error interrupt
	DRV_I2S_INTERRUPT_SOURCE_RECEIVE	Macro to define the Receive interrupt source in case of static driver
	DRV_I2S_INTERRUPT_SOURCE_TRANSMIT	Macro to define the Transmit interrupt source in case of static driver
	DRV_I2S_PERIPHERAL_ID	Configures the I2S PLIB Module ID
	DRV_I2S_RECEIVE_DMA_CHANNEL	Macro to defines the I2S Driver Receive DMA Channel in case of static driver
	DRV_I2S_STOP_IN_IDLE	Identifies whether the driver should stop operations in stop in Idle mode.
	DRV_I2S_TRANSMIT_DMA_CHANNEL	Macro to defines the I2S Driver Transmit DMA Channel in case of static driver
	DRV_I2S_RECEIVE_DMA_CHAINING_CHANNEL	Macro to defines the I2S Driver Receive DMA Chaining Channel in case of static driver

Description

The configuration of the I2S Driver Library is based on the file `sys_config.h`.

This header file contains the configuration selection for the I2S Driver Library. Based on the selections made, the I2S Driver Library may support the selected features. These configuration settings will apply to all instances of the I2S Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

System Configuration

DRV_I2S_INDEX Macro

I2S Static Index selection

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INDEX
```

Description

Index - Used for static drivers

I2S Static Index selection for the driver object reference. This macro defines the driver index in case of static and static multi-client build. For example, if this macro is set to DRV_I2S_INDEX_2, then static driver APIs would be DRV_I2S2_Initialize(), DRV_I2S2_Open() etc. When building static drivers, this macro should be different for each static build of the I2S driver that needs to be included in the project.

Remarks

This index is required to make a reference to the driver object

DRV_I2S_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INSTANCES_NUMBER
```

Description

I2S driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of I2S modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None

DRV_I2S_INTERRUPT_MODE Macro

Macro controls interrupt based operation of the driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INTERRUPT_MODE
```

Description

I2S Interrupt Mode Operation Control

This macro controls the interrupt based operation of the driver. The possible values it can take are

- true - Enables the interrupt mode
- false - Enables the polling mode

If the macro value is true, then Interrupt Service Routine for the interrupt should be defined in the application. The DRV_I2S_Tasks() routine should be called in the ISR.

DRV_I2S_INTERRUPT_SOURCE_ERROR Macro

Defines the interrupt source for the error interrupt

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INTERRUPT_SOURCE_ERROR
```

Description

Error Interrupt Source

Macro to define the Error interrupt source in case of static driver. The interrupt source defined by this macro will override the errorInterruptSource member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the I2S module error interrupt enumeration in the Interrupt PLIB for the microcontroller.

DRV_I2S_INTERRUPT_SOURCE_RECEIVE Macro

Macro to define the Receive interrupt source in case of static driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INTERRUPT_SOURCE_RECEIVE
```

Description

Receive Interrupt Source

Macro to define the Receive interrupt source in case of static driver. The interrupt source defined by this macro will override the rxInterruptSource member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the I2S module receive interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_I2S_INTERRUPT_SOURCE_TRANSMIT Macro

Macro to define the Transmit interrupt source in case of static driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INTERRUPT_SOURCE_TRANSMIT
```

Description

Transmit Interrupt Source

Macro to define the TX interrupt source in case of static driver. The interrupt source defined by this macro will override the txInterruptSource member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the I2S module transmit interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_I2S_PERIPHERAL_ID Macro

Configures the I2S PLIB Module ID

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_PERIPHERAL_ID
```

Description

I2S Peripheral Library Module ID

This macro configures the PLIB ID if the driver is built statically. This value will override the I2SID member of the DRV_I2S_INIT initialization data structure. In that when the driver is built statically, the I2SID member of the DRV_I2S_INIT data structure will be ignored by the driver initialization routine and this macro will be considered. This should be set to the PLIB ID of I2S module (I2S_ID_1, I2S_ID_2 and so on).

DRV_I2S_RECEIVE_DMA_CHANNEL Macro

Macro to defines the I2S Driver Receive DMA Channel in case of static driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_RECEIVE_DMA_CHANNEL
```

Description

I2S Driver Receive DMA Channel

Macro to define the I2S Receive DMA Channel in case of static driver. The DMA channel defined by this macro will override the dmaChannelReceive member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel in the DMA PLIB for the microcontroller.

Remarks

None.

DRV_I2S_STOP_IN_IDLE Macro

Identifies whether the driver should stop operations in stop in Idle mode.

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_STOP_IN_IDLE
```

Description

I2S driver objects configuration

Identifies whether the driver should stop operations in stop in Idle mode. true - Indicates stop in idle mode. false - Indicates do not stop in Idle mode.

Remarks

None

DRV_I2S_TRANSMIT_DMA_CHANNEL Macro

Macro to defines the I2S Driver Transmit DMA Channel in case of static driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_TRANSMIT_DMA_CHANNEL
```

Description

I2S Driver Transmit DMA Channel

Macro to define the I2S Transmit DMA Channel in case of static driver. The DMA channel defined by this macro will override the dmaChannelTransmit member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel in the DMA PLIB for the microcontroller.

Remarks

None.

DRV_I2S_RECEIVE_DMA_CHAINING_CHANNEL Macro

Macro to defines the I2S Driver Receive DMA Chaining Channel in case of static driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_RECEIVE_DMA_CHAINING_CHANNEL
```

Description

I2S Driver Receive DMA Chaining Channel

Macro to define the I2S Receive DMA Chaining Channel in case of static driver. The DMA channel defined by this macro will override the dmaChaningChannelReceive member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel in the DMA PLIB for the microcontroller.

Remarks

None.

Client Configuration

DRV_I2S_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_CLIENTS_NUMBER
```

Description

I2S Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. This value represents the total number of clients to be supported across all hardware instances. So if I2S1 will be accessed by 2 clients and I2S2 will accessed by 3 clients, then this number should be 5. It is recommended that this be set exactly equal to the number of expected clients. Client support consumes RAM memory space. If this macro is not defined and the DRV_I2S_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - single client operation. If this macro is defined and the DRV_I2S_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - multi client operation.

Remarks

None

DRV_I2S_QUEUE_DEPTH_COMBINED Macro

Number of entries of all queues in all instances of the driver.

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_QUEUE_DEPTH_COMBINED
```

Description

I2S Driver Buffer Queue Entries

This macro defined the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for transmit and receive operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build). The hardware instance transmit buffer queue will queue transmit buffers submitted by the DRV_I2S_BufferAddWrite() function. The hardware instance receive buffer queue will queue receive buffers submitted by the DRV_I2S_BufferAddRead() function.

A buffer queue will contains buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all I2S driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking read and write requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its transmit and receive buffer queue size.

As an example, consider the case of static single client driver application where full duplex non blocking operation is desired without queuing, the minimum transmit queue depth and minimum receive queue depth should be 1. Hence the total number of buffer entries should be 2.

As an example, consider the case of a dynamic driver (say 2 instances) where instance 1 will queue up to 3 write requests and up to 2 read requests, and instance 2 will queue up to 2 write requests and up to 6 read requests, the value of this macro should be 13 (2 + 3 + 2 + 6).

Remarks

The maximum combined queue depth should not be greater than 0xFFFF(ie 65535)

Building the Library

This section lists the files that are available in the I2S Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/i2s`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_i2s.h`	This file provides the interface definitions of the I2S driver.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_i2s_dma.c`	This file contains the core implementation of the I2S driver with DMA support.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
`/src/dynamic/drv_i2s_dma_advanced.c`	This file contains the implementation of the I2S driver with DMA support using the channel chaining feature.
`/src/dynamic/drv_i2s.c`	This file contains the implementation of the I2S driver without DMA support.
`/src/dynamic/drv_i2s_read_write.c`	This file contains the basic read/write implementation of the I2S driver.

Module Dependencies

The I2S Driver Library depends on the following modules:

- SPI Peripheral Library
- DMA Peripheral Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_I2S_Deinitialize	Deinitializes the specified instance of the I2S driver module. **Implementation:** Dynamic
	DRV_I2S_Initialize	Initializes hardware and data for the instance of the I2S module. **Implementation:** Dynamic
	DRV_I2S_Status	Gets the current status of the I2S driver module. **Implementation:** Dynamic
	DRV_I2S_Tasks	Maintains the driver's receive state machine and implements its ISR. **Implementation:** Dynamic
	DRV_I2S_TasksError	Maintains the driver's error state machine and implements its ISR. **Implementation:** Dynamic

b) Client Setup Functions

	Name	Description
	DRV_I2S_Close	Closes an opened-instance of the I2S driver. **Implementation:** Dynamic
	DRV_I2S_Open	Opens the specified I2S driver instance and returns a handle to it. **Implementation:** Dynamic

c) Data Transfer Functions

	Name	Description
	DRV_I2S_BufferAddRead	Schedule a non-blocking driver read operation. **Implementation:** Dynamic
	DRV_I2S_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_I2S_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_I2S_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_I2S_BufferProcessedSizeGet	This function returns number of bytes that have been processed for the specified buffer. **Implementation:** Dynamic
	DRV_I2S_BufferCombinedQueueSizeGet	This function returns the number of bytes queued (to be processed) in the buffer queue. **Implementation:** Dynamic
	DRV_I2S_Read	Reads data from the I2S. **Implementation:** Dynamic
	DRV_I2S_Write	Writes data to the I2S. **Implementation:** Dynamic

d) Miscellaneous Functions

	Name	Description
	DRV_I2S_BaudSet	This function sets the baud. **Implementation:** Dynamic

	DRV_I2S_ErrorGet	This function returns the error(if any) associated with the last client request. **Implementation:** Dynamic
	DRV_I2S_ReceiveErrorIgnore	This function enable/disable ignoring of the receive overflow error. **Implementation:** Dynamic
	DRV_I2S_TransmitErrorIgnore	This function enable/disable ignoring of the transmit underrun error. **Implementation:** Dynamic

e) Data Types and Constants

	Name	Description
	DRV_I2S_AUDIO_PROTOCOL_MODE	Identifies the Audio Protocol Mode of the I2S module.
	DRV_I2S_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_I2S_BUFFER_EVENT_HANDLER	Pointer to a I2S Driver Buffer Event handler function
	DRV_I2S_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the driver.
	DRV_I2S_CLOCK_MODE	Identifies the various clock modes of the I2S module.
	DRV_I2S_DATA16	Defines the left and right channel data for 16-bit audio data
	_DRV_I2S_DATA16	Defines the left and right channel data for 16-bit audio data
	DRV_I2S_DATA24	Defines the left and right channel data for 24-bit audio data
	_DRV_I2S_DATA24	Defines the left and right channel data for 24-bit audio data
	DRV_I2S_DATA32	Defines the left and right channel data for 32-bit audio data
	_DRV_I2S_DATA32	Defines the left and right channel data for 32-bit audio data
	DRV_I2S_ERROR	Defines the possible errors that can occur during driver operation.
	DRV_I2S_INIT	Defines the data required to initialize or reinitialize the I2S driver
	DRV_I2S_MODE	Identifies the usage modes of the I2S module.
	DRV_I2S_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_I2S_COUNT	Number of valid I2S driver indices
	DRV_I2S_READ_ERROR	I2S Driver Read Error.
	DRV_I2S_WRITE_ERROR	I2S Driver Write Error.
	DRV_I2S_INDEX_0	I2S driver index definitions
	DRV_I2S_INDEX_1	This is macro DRV_I2S_INDEX_1.
	DRV_I2S_INDEX_2	This is macro DRV_I2S_INDEX_2.
	DRV_I2S_INDEX_3	This is macro DRV_I2S_INDEX_3.
	DRV_I2S_INDEX_4	This is macro DRV_I2S_INDEX_4.
	DRV_I2S_INDEX_5	This is macro DRV_I2S_INDEX_5.

Description

This section describes the Application Programming Interface (API) functions of the I2S Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_I2S_Deinitialize Function

Deinitializes the specified instance of the I2S driver module.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the I2S driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_I2S_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_I2S_Initialize
SYS_STATUS          status;


DRV_I2S_Deinitialize(object);

status = DRV_I2S_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_I2S_Initialize routine

Function

void DRV_I2S_Deinitialize(SYS_MODULE_OBJ object)

DRV_I2S_Initialize Function

Initializes hardware and data for the instance of the I2S module.

Implementation: Dynamic

File

drv_i2s.h

C

```
SYS_MODULE_OBJ DRV_I2S_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *const
init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the I2S driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the I2S module ID. For example, driver instance 0 can be assigned to I2S2. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_I2S_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other I2S routine is called.

This routine should only be called once during system initialization unless DRV_I2S_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

To Enable the DMA mode of operation the init parameters 'dmaChannelTransmit' /'dmaChannelReceive' must be set to valid DMA channel. When DMA mode of operation is enabled, the normal mode(Usual TX and RX) operation is inhibited. When 'dmaChannelTransmit'/'dmaChannelReceive' are set to valid channel numbers the related DMA interrupt source parameters 'dmaInterruptTransmitSource'/ 'dmaInterruptReceiveSource' must be set with appropriate DMA channel interrupt source.

Preconditions

If DMA mode of operation is intended, SYS_DMA_Initialize should have been called before calling this function.

Example

```
DRV_I2S_INIT                init;
SYS_MODULE_OBJ              objectHandle;

init.moduleInit.value           = SYS_MODULE_POWER_RUN_FULL;
init.spiID                      = SPI_ID_1;
init.usageMode                  = DRV_I2S_MODE_MASTER;
init.baudClock                  = SPI_BAUD_RATE_MCLK_CLOCK;
init.baud                       = 48000;
init.clockMode                  = DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_FALL;
init.audioCommWidth             = SPI_AUDIO_COMMUNICATION_24DATA_32FIFO_32CHANNEL;
init.audioTransmitMode          = SPI_AUDIO_TRANSMIT_STEREO;
init.inputSamplePhase           = SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE;
init.protocolMode               = DRV_I2S_AUDIO_I2S;
init.txInterruptSource          = INT_SOURCE_SPI_1_TRANSMIT;
init.rxInterruptSource          = INT_SOURCE_SPI_1_RECEIVE;
init.errorInterruptSource       = INT_SOURCE_SPI_1_ERROR;
init.queueSizeTransmit          = 3;
init.queueSizeReceive           = 2;
init.dmaChannelTransmit         = DMA_CHANNEL_NONE;
init.dmaChannelReceive          = DMA_CHANNEL_NONE;

objectHandle = DRV_I2S_Initialize(DRV_I2S_INDEX_1, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
```

```
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_I2S_Initialize (const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT *const init)

DRV_I2S_Status Function

Gets the current status of the I2S driver module.

Implementation: Dynamic

File

drv_i2s.h

C

```
SYS_STATUS DRV_I2S_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This routine provides the current status of the I2S driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_I2S_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_I2S_Initialize
SYS_STATUS          i2sStatus;

i2sStatus = DRV_I2S_Status(object);
if (SYS_STATUS_READY == i2sStatus)
{
    // This means the driver can be opened using the
    // DRV_I2S_Open( ) function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_I2S_Initialize routine

Function

SYS_STATUS DRV_I2S_Status(SYS_MODULE_OBJ object)

DRV_I2S_Tasks Function

Maintains the driver's receive state machine and implements its ISR.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal receive state machine and implement its transmit and receive ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks function. In interrupt mode, this function should be called from the interrupt service routine of the I2S that is associated with this I2S driver hardware instance. In DMA mode of operation, this function should be called from the interrupt service routine of the channel associated with the transmission/reception of the I2s driver hardware instance.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_I2S_Initialize

while (true)
{
    DRV_I2S_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_I2S_Initialize)

Function

void DRV_I2S_Tasks(SYS_MODULE_OBJ object)

DRV_I2S_TasksError Function

Maintains the driver's error state machine and implements its ISR.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_TasksError(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal error state machine and implement its error ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks() function. In interrupt mode, this function should be called in the error interrupt service routine of the I2S that is associated with this I2S driver hardware instance. In DMA mode of operation, this function should be called from the interrupt service routine of the channel associated with the transmission/reception of the I2s driver hardware instance.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_I2S_Initialize

while (true)
{
    DRV_I2S_TasksError (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_I2S_Initialize)

Function

void DRV_I2S_TasksError (SYS_MODULE_OBJ object)

b) Client Setup Functions

DRV_I2S_Close Function

Closes an opened-instance of the I2S driver.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_Close(const DRV_HANDLE handle);
```

Returns

- None

Description

This routine closes an opened-instance of the I2S driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_I2S_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_I2S_Open

DRV_I2S_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_I2S_Close(DRV_Handle handle)

DRV_I2S_Open Function

Opens the specified I2S driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_i2s.h

C

```
DRV_HANDLE DRV_I2S_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_I2S_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.

Description

This routine opens the specified I2S driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options additionally affect the behavior of the DRV_I2S_Read() and DRV_I2S_Write() functions. If the ioIntent is DRV_IO_INTENT_NONBLOCKING, then these function will not block even if the required amount of data could not be processed. If the ioIntent is DRV_IO_INTENT_BLOCKING, these functions will block until the required amount of data is processed.

If ioIntent is DRV_IO_INTENT_READ, the client will only be read from the driver. If ioIntent is DRV_IO_INTENT_WRITE, the client will only be able to write to the driver. If the ioIntent in DRV_IO_INTENT_READWRITE, the client will be able to do both, read and write.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_I2S_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

Function DRV_I2S_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_I2S_Open(DRV_I2S_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

```
DRV_HANDLE DRV_I2S_Open( const SYS_MODULE_INDEX drvIndex,
const                     DRV_IO_INTENT    ioIntent )
```

c) Data Transfer Functions

DRV_I2S_BufferAddRead Function

Schedule a non-blocking driver read operation.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_BufferAddRead(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE * bufferHandle, void
* buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_I2S_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_I2S_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for write-only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2S driver instance. It should not otherwise be called directly in an ISR.

This function supports DMA mode of operation.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S device instance and the DRV_I2S_Status must have returned SYS_STATUS_READY.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_I2S_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver

DRV_I2S_BufferEventHandlerSet(myI2SHandle,
                APP_I2SBufferEventHandler, (uintptr_t)&myAppObj);

DRV_I2S_BufferAddRead(myI2Shandle, &bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);
```

```
if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the I2S instance as returned by the DRV_I2S_Open function
buffer	Buffer where the received data will be stored.
size	Buffer size in bytes
bufferHandle	Pointer to an argument that will contain the return buffer handle

Function

void DRV_I2S_BufferAddRead(const DRV_HANDLE handle,
DRV_I2S_BUFFER_HANDLE *bufferHandle,
void * buffer, size_t size)

DRV_I2S_BufferAddWrite Function

Schedule a non-blocking driver write operation.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_BufferAddWrite(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE * bufferHandle, void
* buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_I2S_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_I2S_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read-only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2S driver instance. It should not otherwise be called directly in an ISR.

This function supports DMA mode of operation.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S device instance and the DRV_I2S_Status must have returned SYS_STATUS_READY.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_I2S_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver

DRV_I2S_BufferEventHandlerSet(myI2SHandle,
                APP_I2SBufferEventHandler, (uintptr_t)&myAppObj);

DRV_I2S_BufferAddWrite(myI2Shandle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);
```

```
if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the I2S instance as return by the DRV_I2S_Open function
buffer	Data to be transmitted
size	Buffer size in bytes
bufferHandle	Pointer to an argument that will contain the return buffer handle

Function

void DRV_I2S_BufferAddWrite(const DRV_HANDLE handle,
DRV_I2S_BUFFER_HANDLE *bufferHandle,
void * buffer, size_t size);

DRV_I2S_BufferAddWriteRead Function

Schedule a non-blocking driver write-read operation.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_BufferAddWriteRead(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE * bufferHandle,
void * transmitBuffer, void * receiveBuffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_I2S_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_I2S_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2S driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every I2S write. The transmit and receive size must be same.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S device instance and the DRV_I2S_Status must have returned SYS_STATUS_READY.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_I2S_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver

DRV_I2S_BufferEventHandlerSet(myI2SHandle,
                APP_I2SBufferEventHandler, (uintptr_t)&myAppObj);
```

```
DRV_I2S_BufferAddWriteRead(myI2Shandle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the I2S instance as returned by the DRV_I2S_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	Buffer where the transmit data will be stored
receiveBuffer	Buffer where the received data will be stored
size	Buffer size in bytes

Function

void DRV_I2S_BufferAddWriteRead(const DRV_HANDLE handle,

DRV_I2S_BUFFER_HANDLE *bufferHandle,

void *transmitBuffer, void *receiveBuffer,

size_t size)

DRV_I2S_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_I2S_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls either the DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite or DRV_I2S_BufferAddWriteRead function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandler,
                                (uintptr_t)&myAppObj);

DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.
```

```
void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE handle, uintptr_t contextHandle)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_I2S_BufferEventHandlerSet(const DRV_HANDLE handle,
DRV_I2S_BUFFER_EVENT_HANDLER eventHandler,
uintptr_t contextHandle)

DRV_I2S_BufferProcessedSizeGet Function

This function returns number of bytes that have been processed for the specified buffer.

Implementation: Dynamic

File

drv_i2s.h

C

```
size_t DRV_I2S_BufferProcessedSizeGet(DRV_I2S_BUFFER_HANDLE bufferHandle);
```

Returns

Returns the number of the bytes that have been processed for this buffer. Returns 0 for an invalid or an expired buffer handle.

Description

This function returns number of bytes that have been processed for the specified buffer. The client can use this function, in a case where the buffer has terminated due to an error, to obtain the number of bytes that have been processed. If this function is called on a invalid buffer handle, or if the buffer handle has expired, the function returns 0.

Remarks

None.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

One of DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite or DRV_I2S_BufferAddWriteRead function must have been called and a valid buffer handle returned.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandle,
                                                    (uintptr_t)&myAppObj);

DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
```

```
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred.

            processedBytes = DRV_I2S_BufferProcessedSizeGet(bufferHandle);

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
bufferhandle	Handle of the buffer of which the processed number of bytes to be obtained.

Function

size_t DRV_I2S_BufferProcessedSizeGet(DRV_I2S_BUFFER_HANDLE bufferHandle)

DRV_I2S_BufferCombinedQueueSizeGet Function

This function returns the number of bytes queued (to be processed) in the buffer queue.

Implementation: Dynamic

File

drv_i2s.h

C

```
size_t DRV_I2S_BufferCombinedQueueSizeGet(DRV_HANDLE handle);
```

Returns

Returns the number of the bytes that have been processed for this buffer. Returns 0 for an invalid or an expired client handle.

Description

This function returns the number of bytes queued (to be processed) in the buffer queue associated with the driver instance to which the calling client belongs. The client can use this function to know number of bytes that is in the queue to be transmitted.

Remarks

None.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

One of DRV_I2S_BufferAddRead/DRV_I2S_BufferAddWrite function must have been called and buffers should have been queued for transmission.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
size_t bufferQueuedSize;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandle,
                                                        (uintptr_t)&myAppObj);

DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// The data is being processed after adding the buffer to the queue.
// The user can get to know dynamically available data in the queue to be
// transmitted by calling DRV_I2S_BufferCombinedQueueSizeGet
bufferQueuedSize = DRV_I2S_BufferCombinedQueueSizeGet(myI2SHandle);
```

Parameters

Parameters	Description
handle	Opened client handle associated with a driver object.

Function

size_t DRV_I2S_BufferCombinedQueueSizeGet(DRV_HANDLE handle)

DRV_I2S_Read Function

Reads data from the I2S.

Implementation: Dynamic

File

drv_i2s.h

C

```
size_t DRV_I2S_Read(const DRV_HANDLE handle, uint8_t * buffer, const size_t numBytes);
```

Returns

Number of bytes actually copied into the caller's buffer. Returns DRV_I2S_READ_ERROR in case of an error.

Description

This routine reads data from the I2S. This function is blocking if the driver was opened by the client for blocking operation. This function will not block if the driver was opened by the client for non blocking operation. If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_BLOCKING, this function will only return when (or will block until) numBytes of bytes have been received or if an error occurred.

If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_NON_BLOCKING, this function will return with the number of bytes that were actually read. The function will not wait until numBytes of bytes have been read.

Remarks

This function is thread safe in a RTOS application. It is recommended that this function not be called in I2S Driver Event Handler due to the potential blocking nature of the function. This function should not be called directly in an ISR. It should not be called in the event handler associated with another I2S driver instance.

This function does not supports DMA mode of operation.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_I2S_Open call.

Example

```
DRV_HANDLE      myI2SHandle;    // Returned from DRV_I2S_Open
char            myBuffer[MY_BUFFER_SIZE];
unsigned int    count;
unsigned int    total;

total = 0;
do
{
    count  = DRV_I2S_Read(myI2SHandle, &myBuffer[total],
                               MY_BUFFER_SIZE - total);
    total += count;
    if(count == DRV_I2S_READ_ERROR)
    {
        // Handle error ...
    }
    else
    {
        // Do what needs to be..
    }
} while( total < MY_BUFFER_SIZE );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
buffer	Buffer into which the data read from the I2S instance will be placed.
numBytes	Total number of bytes that need to be read from the module instance (must be equal to or less than the size of the buffer)

Function

size_t DRV_I2S_Read(const DRV_HANDLE handle, uint8_t *buffer,
const size_t numBytes)

DRV_I2S_Write Function

Writes data to the I2S.

Implementation: Dynamic

File

drv_i2s.h

C

```
size_t DRV_I2S_Write(const DRV_HANDLE handle, uint8_t * buffer, const size_t numBytes);
```

Returns

Number of bytes actually written to the driver. Return DRV_I2S_WRITE_ERROR in case of an error.

Description

This routine writes data to the I2S. This function is blocking if the driver was opened by the client for blocking operation. This function will not block if the driver was opened by the client for non blocking operation. If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_BLOCKING, this function will only return when (or will block until) numbytes of bytes have been transmitted or if an error occurred.

If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_NON_BLOCKING, this function will return with the number of bytes that were actually accepted for transmission. The function will not wait until numBytes of bytes have been transmitted.

Remarks

This function is thread safe in a RTOS application. It is recommended that this function not be called in I2S Driver Event Handler due to the potential blocking nature of the function. This function should not be called directly in an ISR. It should not be called in the event handler associated with another USART driver instance.

This function does not supports DMA mode of operation.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_I2S_Open call.

Example

```
DRV_HANDLE myI2SHandle;    // Returned from DRV_I2S_Open
char myBuffer[MY_BUFFER_SIZE];
int count;
unsigned int total;
total = 0;
do
{
    count  = DRV_I2S_Write(myI2SHandle, &myBuffer[total],
            MY_BUFFER_SIZE - total);
    total += count;
    if(count == DRV_I2S_WRITE_ERROR)
    {
        // Handle error ...
    }
    else
    {
        // Do what needs to be ..
    }
} while( total < MY_BUFFER_SIZE );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

buffer	Buffer containing the data to written.
numbytes	size of the buffer

Function

size_t DRV_I2S_Write(const DRV_HANDLE handle, void * buffer,
const size_t numbytes)

d) Miscellaneous Functions

DRV_I2S_BaudSet Function

This function sets the baud.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_BaudSet(DRV_HANDLE handle, uint32_t spiClock, uint32_t baud);
```

Returns

None

Description

This function sets the baud rate for the I2S operation.

Remarks

None.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_HANDLE handle;
uint32_t baud;
uint32_t baud;*

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandle,
                                                  (uintptr_t)&myAppObj);

// Sets the baud rate to a new value as below
baud = 115200;
DRV_I2S_BaudSet(myI2SHandle,baud)

// Further perform the operation needed
DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
```

```
    size_t processedBytes;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred.

            processedBytes = DRV_I2S_BufferProcessedSizeGet(bufferHandle);

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
i2sClock	The Source clock frequency to the i2S module.
baud	The baud to be set.

Function

void DRV_I2S_BaudSet(DRV_HANDLE handle, uint32_t baud)

DRV_I2S_ErrorGet Function

This function returns the error(if any) associated with the last client request.

Implementation: Dynamic

File

drv_i2s.h

C

```
DRV_I2S_ERROR DRV_I2S_ErrorGet(DRV_HANDLE handle);
```

Returns

A DRV_I2S_ERROR type indicating last known error status.

Description

This function returns the error(if any) associated with the last client request. The DRV_I2S_Read() and DRV_I2S_Write() will update the client error status when these functions return DRV_I2S_READ_ERROR and DRV_I2S_WRITE_ERROR, respectively. If the driver send a DRV_I2S_BUFFER_EVENT_ERROR to the client, the client can call this function to know the error cause. The error status will be updated on every operation and should be read frequently (ideally immediately after the driver operation has completed) to know the relevant error status.

Remarks

It is the client's responsibility to make sure that the error status is obtained frequently. The driver will update the client error status regardless of whether this has been examined by the client.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet( myI2SHandle, APP_I2SBufferEventHandle,
                                 (uintptr_t)&myAppObj );

DRV_I2S_BufferAddRead( myI2Shandle,&bufferHandle,
                                        myBuffer, MY_BUFFER_SIZE );

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler( DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle )
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;
```

```
    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_SUCCESS:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_FAILURE:

            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred. We can also find
            // the error cause.

            processedBytes = DRV_I2S_BufferProcessedSizeGet(bufferHandle);
            if(DRV_I2S_ERROR_RECEIVE_OVERRUN == DRV_I2S_ErrorGet(myI2SHandle))
            {
                // There was an receive over flow error.
                // Do error handling here.
            }

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
bufferhandle	Handle of the buffer of which the processed number of bytes to be obtained.

Function

DRV_I2S_ERROR DRV_I2S_ErrorGet(DRV_HANDLE handle)

DRV_I2S_ReceiveErrorIgnore Function

This function enable/disable ignoring of the receive overflow error.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_ReceiveErrorIgnore(DRV_HANDLE handle, bool errorEnable);
```

Returns

None

Description

A receive overflow is not a critical error; during receive overflow data in the FIFO is not overwritten by receive data. Ignore receive overflow is needed for cases when there is a general performance problem in the system that software must handle properly.

Remarks

None.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_HANDLE handle;
uint32_t baud;
uint32_t baud;*

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandle,
                                                        (uintptr_t)&myAppObj);

// Enable ignoring of receive overflow error
DRV_I2S_ReceiveErrorIgnore(myI2SHandle, true);

// Further perform the operation needed
DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // The context handle was set to an application specific
```

```
        // object. It is now retrievable easily in the event handler.
        MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
        size_t processedBytes;

        switch(event)
        {
            case DRV_I2S_BUFFER_EVENT_COMPLETE:

                // This means the data was transferred.
                break;

            case DRV_I2S_BUFFER_EVENT_ERROR:

                // Error handling here.
                // We can find out how many bytes were processed in this
                // buffer before the error occurred.

                processedBytes = DRV_I2S_BufferProcessedSizeGet(bufferHandle);

                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
errorIgnore	When set to 'true' enables ignoring of transmit underrun error. When set to 'false' disables ignoring of transmit underrun error.

Function

void DRV_I2S_ReceiveErrorIgnore(DRV_HANDLE handle, bool errorEnable)

DRV_I2S_TransmitErrorIgnore Function

This function enable/disable ignoring of the transmit underrun error.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_TransmitErrorIgnore(DRV_HANDLE handle, bool errorIgnore);
```

Returns

None

Description

A Transmit underrun error is not a critical error and zeros are transmitted until the SPIxTXB is not empty. Ignore Transmit underrun error is needed for cases when software does not care or does not need to know about underrun conditions.

Remarks

None.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_HANDLE handle;
uint32_t baud;
uint32_t baud;*

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandle,
                                                        (uintptr_t)&myAppObj);

// Enable ignoring of transmit underrun error
DRV_I2S_TransmitErrorIgnore(myI2SHandle, true);

// Further perform the operation needed
DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // The context handle was set to an application specific
```

```
        // object. It is now retrievable easily in the event handler.
        MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
        size_t processedBytes;

        switch(event)
        {
            case DRV_I2S_BUFFER_EVENT_COMPLETE:

                // This means the data was transferred.
                break;

            case DRV_I2S_BUFFER_EVENT_ERROR:

                // Error handling here.
                // We can find out how many bytes were processed in this
                // buffer before the error occurred.

                processedBytes = DRV_I2S_BufferProcessedSizeGet(bufferHandle);

                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
errorIgnore	When set to 'true' enables ignoring of transmit underrun error. When set to 'false' disables ignoring of transmit underrun error.

Function

void DRV_I2S_TransmitErrorIgnore(DRV_HANDLE handle, bool errorIgnore)

e) Data Types and Constants

DRV_I2S_AUDIO_PROTOCOL_MODE Enumeration

Identifies the Audio Protocol Mode of the I2S module.

File

drv_i2s.h

C

```
typedef enum {
  DRV_I2S_AUDIO_I2S,
  DRV_I2S_AUDIO_LFET_JUSTIFIED,
  DRV_I2S_AUDIO_RIGHT_JUSTIFIED,
  DRV_I2S_AUDIO_PCM_DSP
} DRV_I2S_AUDIO_PROTOCOL_MODE;
```

Members

Members	Description
DRV_I2S_AUDIO_I2S	I2S Audio Protocol
DRV_I2S_AUDIO_LFET_JUSTIFIED	Left Justified Audio Protocol
DRV_I2S_AUDIO_RIGHT_JUSTIFIED	Right Justified Audio Protocol
DRV_I2S_AUDIO_PCM_DSP	PCM/DSP Audio Protocol

Description

I2S Audio Protocol Mode

This enumeration identifies Audio Protocol Mode of the I2S module.

Remarks

None.

DRV_I2S_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_i2s.h

C

```
typedef enum {
  DRV_I2S_BUFFER_EVENT_COMPLETE,
  DRV_I2S_BUFFER_EVENT_ERROR,
  DRV_I2S_BUFFER_EVENT_ABORT
} DRV_I2S_BUFFER_EVENT;
```

Members

Members	Description
DRV_I2S_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_I2S_BUFFER_EVENT_ERROR	Error while processing the request
DRV_I2S_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

I2S Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite or DRV_I2S_BufferAddWriteRead functions.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_I2S_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_I2S_BUFFER_EVENT_HANDLER Type

Pointer to a I2S Driver Buffer Event handler function

File

drv_i2s.h

C

```
typedef void (* DRV_I2S_BUFFER_EVENT_HANDLER)(DRV_I2S_BUFFER_EVENT event, DRV_I2S_BUFFER_HANDLE
bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

I2S Driver Buffer Event Handler Function

This data type defines the required function signature for the I2S driver buffer event handling callback function. A client must register a pointer to a buffer event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_I2S_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_I2S_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_I2S_ErrorGet function can be called to know the error. The DRV_I2S_BufferProcessedSizeGet function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_I2S_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite and DRV_I2S_BufferAddWriteRead functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running. For example, buffers cannot be added I2S2 driver in I2S1 driver event handler.

Example

```
void APP_MyBufferEventHandler( DRV_I2S_BUFFER_EVENT event,
                               DRV_I2S_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // Handle the completed buffer.
            break;
```

```
        case DRV_I2S_BUFFER_EVENT_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_I2S_BUFFER_HANDLE Type

Handle identifying a read or write buffer passed to the driver.

File

drv_i2s.h

C

```
typedef uintptr_t DRV_I2S_BUFFER_HANDLE;
```

Description

I2S Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite, and DRV_I2S_BufferAddWriteRead functions. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_I2S_CLOCK_MODE Enumeration

Identifies the various clock modes of the I2S module.

File

drv_i2s.h

C

```
typedef enum {
  DRV_I2S_CLOCK_MODE_IDLE_LOW_EDGE_RISE,
  DRV_I2S_CLOCK_MODE_IDLE_LOW_EDGE_FALL,
  DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_FALL,
  DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_RISE
} DRV_I2S_CLOCK_MODE;
```

Members

Members	Description
DRV_I2S_CLOCK_MODE_IDLE_LOW_EDGE_RISE	I2S Clock Mode 0 - Idle State Low & Sampling on Rising Edge
DRV_I2S_CLOCK_MODE_IDLE_LOW_EDGE_FALL	I2S Clock Mode 1 - Idle State Low & Sampling on Falling Edge
DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_FALL	I2S Clock Mode 2 - Idle State High & Sampling on Falling Edge
DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_RISE	I2S Clock Mode 3 - Idle State High & Sampling on Rising Edge

Description

I2S Clock Mode Selection

This enumeration identifies the supported clock modes of the I2S module.

Remarks

None.

DRV_I2S_DATA16 Structure

Defines the left and right channel data for 16-bit audio data

File

drv_i2s.h

C

```
typedef struct _DRV_I2S_DATA16 {
  int16_t leftData;
  int16_t rightData;
} DRV_I2S_DATA16;
```

Members

Members	Description
int16_t leftData;	Left channel data
int16_t rightData;	Right channel data

Description

I2S Driver Audio Data 16

Defines the left and right channel data for 16-bit audio data

Remarks

None.

DRV_I2S_DATA24 Structure

Defines the left and right channel data for 24-bit audio data

File

drv_i2s.h

C

```
typedef struct _DRV_I2S_DATA24 {
  int32_t leftData : 24;
  int32_t leftDataPad : 8;
  int32_t rightData : 24;
  int32_t rightDataPad : 8;
} DRV_I2S_DATA24;
```

Members

Members	Description
int32_t leftData : 24;	Left channel data
int32_t leftDataPad : 8;	Left channel data pad
int32_t rightData : 24;	Right channel data
int32_t rightDataPad : 8;	Right channel data pad

Description

I2S Driver Audio Data 24

Defines the left and right channel data for 24-bit audio data

Remarks

None.

DRV_I2S_DATA32 Structure

Defines the left and right channel data for 32-bit audio data

File

drv_i2s.h

C

```
typedef struct _DRV_I2S_DATA32 {
  int32_t leftData;
  int32_t rightDataPad;
} DRV_I2S_DATA32;
```

Members

Members	Description
int32_t leftData;	Left channel data
int32_t rightDataPad;	Right channel data

Description

I2S Driver Audio Data 32

Defines the left and right channel data for 32-bit audio data

Remarks

None.

DRV_I2S_ERROR Enumeration

Defines the possible errors that can occur during driver operation.

File

drv_i2s.h

C

```
typedef enum {
  DRV_I2S_ERROR_NONE,
  DRV_I2S_ERROR_RECEIVE_OVERFLOW,
  DRV_I2S_ERROR_TRANSMIT_UNDERUN,
  DRV_I2S_ERROR_FRAMING,
  DRV_I2S_ERROR_ADDRESS
} DRV_I2S_ERROR;
```

Members

Members	Description
DRV_I2S_ERROR_NONE	Data was transferred successfully.
DRV_I2S_ERROR_RECEIVE_OVERFLOW	Receive overflow error.
DRV_I2S_ERROR_TRANSMIT_UNDERUN	Transmit underrun error.
DRV_I2S_ERROR_FRAMING	Framing error.
DRV_I2S_ERROR_ADDRESS	Channel address error (Applicable in DMA mode)

Description

I2S Driver Error

This data type defines the possible errors that can occur when occur during USART driver operation. These values are returned by DRV_I2S_ErrorGet function.

Remarks

None.

DRV_I2S_INIT Structure

Defines the data required to initialize or reinitialize the I2S driver

File

drv_i2s.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SPI_MODULE_ID spiID;
  DRV_I2S_MODE usageMode;
  SPI_BAUD_RATE_CLOCK baudClock;
  uint32_t baud;
  DRV_I2S_CLOCK_MODE clockMode;
  SPI_AUDIO_COMMUNICATION_WIDTH audioCommWidth;
  SPI_AUDIO_TRANSMIT_MODE audioTransmitMode;
  SPI_INPUT_SAMPLING_PHASE inputSamplePhase;
  DRV_I2S_AUDIO_PROTOCOL_MODE protocolMode;
  INT_SOURCE txInterruptSource;
  INT_SOURCE rxInterruptSource;
  INT_SOURCE errorInterruptSource;
  uint32_t queueSizeTransmit;
  uint32_t queueSizeReceive;
  DMA_CHANNEL dmaChannelTransmit;
  DMA_CHANNEL dmaChaningChannelTransmit;
  DMA_CHANNEL dmaChannelReceive;
  DMA_CHANNEL dmaChaningChannelReceive;
  INT_SOURCE dmaInterruptTransmitSource;
  INT_SOURCE dmaInterruptChainingTransmitSource;
  INT_SOURCE dmaInterruptReceiveSource;
  INT_SOURCE dmaInterruptChainingReceiveSource;
} DRV_I2S_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SPI_MODULE_ID spiID;	Identifies I2S hardware module ID(PLIB-level SPI ID) For static build of the driver, this is overridden by DRV_I2S_MODULE_ID macro in system_config.h.
DRV_I2S_MODE usageMode;	Usage Mode Type
SPI_BAUD_RATE_CLOCK baudClock;	Select the clock which generates the baud rate The options available are Peripheral Clock/ Reference clock.
uint32_t baud;	Initial Baud Rate Value
DRV_I2S_CLOCK_MODE clockMode;	Clock mode
SPI_AUDIO_COMMUNICATION_WIDTH audioCommWidth;	Audio communication width
SPI_AUDIO_TRANSMIT_MODE audioTransmitMode;	Audio mono/stereo
SPI_INPUT_SAMPLING_PHASE inputSamplePhase;	Input Sample Phase Selection
DRV_I2S_AUDIO_PROTOCOL_MODE protocolMode;	Audio Protocol Mode
INT_SOURCE txInterruptSource;	Interrupt Source for Transmit Interrupt. For static build of the driver, this is overridden by the DRV_I2S_TRANSMIT_INTERRUPT_SOURCE macro in system_config.h.
INT_SOURCE rxInterruptSource;	Interrupt Source for Receive Interrupt. For static build of the driver, this is overridden by the DRV_I2S_RECEIVE_INTERRUPT_SOURCE macro in system_config.h.

INT_SOURCE errorInterruptSource;	Interrupt Source for Error Interrupt. For static build of the driver, this is overridden by the DRV_I2S_ERROR_INTERRUPT_SOURCE macro in system_config.h.
uint32_t queueSizeTransmit;	This is the transmit buffer queue size. This is the maximum number of write requests that driver will queue. For a static build of the driver, this is overridden by the DRV_I2S_TRANSMIT_QUEUE_SIZE macro in system_config.h.
uint32_t queueSizeReceive;	This is the receive buffer queue size. This is the maximum number of read requests that driver will queue. For a static build of the driver, this is overridden by the DRV_I2S_RECEIVE_QUEUE_SIZE macro in system_config.h.
DMA_CHANNEL dmaChannelTransmit;	This is the transmit DMA channel. A value of DMA_CHANNEL_NONE indicates DMA is not required for Tx. For a static build of the driver this is overridden by the DRV_I2S_TRANSMIT_DMA_CHANNEL macro in system_config.h.
DMA_CHANNEL dmaChaningChannelTransmit;	This is the transmit DMA chaining channel. This channel is needed when DMA is needed to be used channel chaining mode for Tx. Channel Chaining could be used to obtain high quality/resolution audio.
DMA_CHANNEL dmaChannelReceive;	This is the receive DMA channel. A value of DMA_CHANNEL_NONE indicates DMA is not required for Rx. For a static build of the driver this is overridden by the DRV_I2S_RECEIVE_DMA_CHANNEL macro in system_config.h.
DMA_CHANNEL dmaChaningChannelReceive;	This is the receive DMA chaining channel. This channel is needed when DMA is needed to be used channel chaining mode for Rx. Channel Chaining could be used to obtain high quality/resolution audio.
INT_SOURCE dmaInterruptTransmitSource;	This is the transmit DMA channel interrupt. This is applicable only if 'dmaChannelTransmit' has a valid channel number. This takes the interrupt source number for the corresponding DMA channel.
INT_SOURCE dmaInterruptChainingTransmitSource;	This is the transmit DMA chaining channel interrupt. This is applicable only if 'dmaChaningChannelTransmit' has a valid channel number. This takes the interrupt source number for the corresponding DMA chaining channel.
INT_SOURCE dmaInterruptReceiveSource;	This is the receive DMA channel interrupt. This is applicable only if 'dmaChannelReceive' has a valid channel number. This takes the interrupt source number for the corresponding DMA channel.
INT_SOURCE dmaInterruptChainingReceiveSource;	This is the receive DMA chaining channel interrupt. This is applicable only if 'dmaInterruptReceiveSource' has a valid channel number. This takes the interrupt source number for the corresponding DMA chaining channel.

Description

I2S Driver Initialization Data

This data type defines the data required to initialize or reinitialize the I2S driver. If the driver is built statically, some members of this data structure are statically over-ridden by static overrides in the system_config.h file.

Remarks

None.

DRV_I2S_MODE Enumeration

Identifies the usage modes of the I2S module.

File

drv_i2s.h

C

```
typedef enum {
  DRV_I2S_MODE_SLAVE,
  DRV_I2S_MODE_MASTER
} DRV_I2S_MODE;
```

Members

Members	Description
DRV_I2S_MODE_SLAVE	I2S Mode Slave
DRV_I2S_MODE_MASTER	I2S Mode Master

Description

I2S Usage Modes Enumeration

This enumeration identifies the whether the I2S module will be used as a master or slave.

Remarks

None.

DRV_I2S_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_i2s.h

C

```
#define DRV_I2S_BUFFER_HANDLE_INVALID ((DRV_I2S_BUFFER_HANDLE)(-1))
```

Description

I2S Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite and DRV_I2S_BufferAddWriteRead functions if the buffer add request was not successful.

Remarks

None

DRV_I2S_COUNT Macro

Number of valid I2S driver indices

File

drv_i2s.h

C

```
#define DRV_I2S_COUNT
```

Description

I2S Driver Module Count

This constant identifies the maximum number of I2S Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of I2S instances on this microcontroller.

Remarks

This value is part-specific.

DRV_I2S_READ_ERROR Macro

I2S Driver Read Error.

File

drv_i2s.h

C

```
#define DRV_I2S_READ_ERROR ((size_t)(-1))
```

Description

I2S Driver Read Error

This constant is returned by DRV_I2S_Read function when an error occurs.

Remarks

None.

DRV_I2S_WRITE_ERROR Macro

I2S Driver Write Error.

File

drv_i2s.h

C

```
#define DRV_I2S_WRITE_ERROR ((size_t)(-1))
```

Description

I2S Driver Write Error

This constant is returned by DRV_I2S_Write() function when an error occurs.

Remarks

None.

DRV_I2S_INDEX_0 Macro

I2S driver index definitions

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_0 0
```

Description

Driver I2S Module Index

These constants provide I2S driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_I2S_Initialize and DRV_I2S_Open routines to identify the driver instance in use.

DRV_I2S_INDEX_1 Macro

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_1 1
```

Description

This is macro DRV_I2S_INDEX_1.

DRV_I2S_INDEX_2 Macro

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_2 2
```

Description

This is macro DRV_I2S_INDEX_2.

DRV_I2S_INDEX_3 Macro

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_3 3
```

Description

This is macro DRV_I2S_INDEX_3.

DRV_I2S_INDEX_4 Macro

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_4 4
```

Description

This is macro DRV_I2S_INDEX_4.

DRV_I2S_INDEX_5 Macro

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_5 5
```

Description

This is macro DRV_I2S_INDEX_5.

Files

Files

Name	Description
drv_i2s.h	I2S Driver Interface header file
drv_i2s_config_template.h	I2S Driver Configuration Template.

Description

drv_i2s.h

I2S Driver Interface header file

Enumerations

	Name	Description
	DRV_I2S_AUDIO_PROTOCOL_MODE	Identifies the Audio Protocol Mode of the I2S module.
	DRV_I2S_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_I2S_CLOCK_MODE	Identifies the various clock modes of the I2S module.
	DRV_I2S_ERROR	Defines the possible errors that can occur during driver operation.
	DRV_I2S_MODE	Identifies the usage modes of the I2S module.

Functions

	Name	Description
	DRV_I2S_BaudSet	This function sets the baud. **Implementation:** Dynamic
	DRV_I2S_BufferAddRead	Schedule a non-blocking driver read operation. **Implementation:** Dynamic
	DRV_I2S_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_I2S_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_I2S_BufferCombinedQueueSizeGet	This function returns the number of bytes queued (to be processed) in the buffer queue. **Implementation:** Dynamic
	DRV_I2S_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_I2S_BufferProcessedSizeGet	This function returns number of bytes that have been processed for the specified buffer. **Implementation:** Dynamic
	DRV_I2S_Close	Closes an opened-instance of the I2S driver. **Implementation:** Dynamic
	DRV_I2S_Deinitialize	Deinitializes the specified instance of the I2S driver module. **Implementation:** Dynamic

	Name	Description
	DRV_I2S_ErrorGet	This function returns the error(if any) associated with the last client request. **Implementation:** Dynamic
	DRV_I2S_Initialize	Initializes hardware and data for the instance of the I2S module. **Implementation:** Dynamic
	DRV_I2S_Open	Opens the specified I2S driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_I2S_Read	Reads data from the I2S. **Implementation:** Dynamic
	DRV_I2S_ReceiveErrorIgnore	This function enable/disable ignoring of the receive overflow error. **Implementation:** Dynamic
	DRV_I2S_Status	Gets the current status of the I2S driver module. **Implementation:** Dynamic
	DRV_I2S_Tasks	Maintains the driver's receive state machine and implements its ISR. **Implementation:** Dynamic
	DRV_I2S_TasksError	Maintains the driver's error state machine and implements its ISR. **Implementation:** Dynamic
	DRV_I2S_TransmitErrorIgnore	This function enable/disable ignoring of the transmit underrun error. **Implementation:** Dynamic
	DRV_I2S_Write	Writes data to the I2S. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_I2S_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_I2S_COUNT	Number of valid I2S driver indices
	DRV_I2S_INDEX_0	I2S driver index definitions
	DRV_I2S_INDEX_1	This is macro DRV_I2S_INDEX_1.
	DRV_I2S_INDEX_2	This is macro DRV_I2S_INDEX_2.
	DRV_I2S_INDEX_3	This is macro DRV_I2S_INDEX_3.
	DRV_I2S_INDEX_4	This is macro DRV_I2S_INDEX_4.
	DRV_I2S_INDEX_5	This is macro DRV_I2S_INDEX_5.
	DRV_I2S_READ_ERROR	I2S Driver Read Error.
	DRV_I2S_WRITE_ERROR	I2S Driver Write Error.

Structures

	Name	Description
	_DRV_I2S_DATA16	Defines the left and right channel data for 16-bit audio data
	_DRV_I2S_DATA24	Defines the left and right channel data for 24-bit audio data
	_DRV_I2S_DATA32	Defines the left and right channel data for 32-bit audio data
	DRV_I2S_DATA16	Defines the left and right channel data for 16-bit audio data
	DRV_I2S_DATA24	Defines the left and right channel data for 24-bit audio data
	DRV_I2S_DATA32	Defines the left and right channel data for 32-bit audio data

	DRV_I2S_INIT	Defines the data required to initialize or reinitialize the I2S driver

Types

	Name	Description
	DRV_I2S_BUFFER_EVENT_HANDLER	Pointer to a I2S Driver Buffer Event handler function
	DRV_I2S_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the driver.

Description

I2S Driver Interface

The I2S device driver provides a simple interface to manage the I2S module on Microchip microcontrollers. This file provides the interface definition for the I2S driver.

File Name

drv_i2s.h

Company

Microchip Technology Inc.

drv_i2s_config_template.h

I2S Driver Configuration Template.

Macros

	Name	Description
	DRV_I2S_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_I2S_INDEX	I2S Static Index selection
	DRV_I2S_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_I2S_INTERRUPT_MODE	Macro controls interrupt based operation of the driver
	DRV_I2S_INTERRUPT_SOURCE_ERROR	Defines the interrupt source for the error interrupt
	DRV_I2S_INTERRUPT_SOURCE_RECEIVE	Macro to define the Receive interrupt source in case of static driver
	DRV_I2S_INTERRUPT_SOURCE_TRANSMIT	Macro to define the Transmit interrupt source in case of static driver
	DRV_I2S_PERIPHERAL_ID	Configures the I2S PLIB Module ID
	DRV_I2S_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.
	DRV_I2S_RECEIVE_DMA_CHAINING_CHANNEL	Macro to defines the I2S Driver Receive DMA Chaining Channel in case of static driver
	DRV_I2S_RECEIVE_DMA_CHANNEL	Macro to defines the I2S Driver Receive DMA Channel in case of static driver
	DRV_I2S_STOP_IN_IDLE	Identifies whether the driver should stop operations in stop in Idle mode.
	DRV_I2S_TRANSMIT_DMA_CHANNEL	Macro to defines the I2S Driver Transmit DMA Channel in case of static driver

Description

I2S Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in

the driver.

File Name

drv_i2s_config_template.h

Company

Microchip Technology Inc.

Input Capture Driver Library

This topic describes the Input Capture Driver Library.

Introduction

The Input Capture Static Driver provides a high-level interface to manage the Input Capture module on the Microchip family of microcontrollers.

Description

Through the MHC, this driver provides APIs for the following:

- Initializing the module
- Starting/Stopping of the capture
- 16/32-bit data reads
- Buffer empty status

Library Interface

Functions

	Name	Description
	DRV_IC_Initialize	Initializes the Input Capture instance for the specified driver index. **Implementation:** Static
	DRV_IC_BufferIsEmpty	Returns the Input Capture instance buffer empty status for the specified driver index. **Implementation:** Static
	DRV_IC_Capture16BitDataRead	Reads the 16-bit Input Capture for the specified driver index. **Implementation:** Static
	DRV_IC_Capture32BitDataRead	Reads the 32-bit Input Capture for the specified driver index. **Implementation:** Static
	DRV_IC_Start	Starts the Input Capture instance for the specified driver index. **Implementation:** Static
	DRV_IC_Stop	Stops the Input Capture instance for the specified driver index. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the Input Capture Driver Library.

Functions

DRV_IC_Initialize Function

Initializes the Input Capture instance for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
void DRV_IC_Initialize();
```

Returns

None.

Description

This routine initializes the Input Capture driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters. The driver instance index is independent of the Input Capture module ID. For example, driver instance 0 can be assigned to Input Capture 2.

Remarks

This routine must be called before any other Input Capture routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_IC_Initialize(void)

DRV_IC_BufferIsEmpty Function

Returns the Input Capture instance buffer empty status for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
bool DRV_IC_BufferIsEmpty();
```

Returns

Boolean

- 1 - Buffer is empty
- 0 - Buffer is not empty

Description

Returns the Input Capture instance buffer empty status for the specified driver index. The function should be called to determine whether or not the IC buffer has data.

Remarks

None.

Preconditions

DRV_IC_Initialize has been called.

Function

bool DRV_IC_BufferIsEmpty(void)

DRV_IC_Capture16BitDataRead Function

Reads the 16-bit Input Capture for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
uint16_t DRV_IC_Capture16BitDataRead();
```

Returns

uint16_t value of the data read from the Input Capture.

Description

This routine reads the 16-bit data for the specified driver index.

Remarks

None.

Preconditions

DRV_IC_Initialize has been called.

Function

uint16_t DRV_IC_Capture16BitDataRead(void)

DRV_IC_Capture32BitDataRead Function

Reads the 32-bit Input Capture for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
uint32_t DRV_IC_Capture32BitDataRead();
```

Returns

uint32_t value of the data read from the Input Capture.

Description

This routine reads the 32-bit data for the specified driver index

Remarks

None.

Preconditions

DRV_IC_Initialize has been called.

Function

uint32_t DRV_IC_Capture32BitDataRead(void)

DRV_IC_Start Function

Starts the Input Capture instance for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
void DRV_IC_Start();
```

Returns

None.

Description

This routine starts the Input Capture driver for the specified driver index, starting an input capture.

Remarks

None.

Preconditions

DRV_IC_Initialize has been called.

Function

void DRV_IC_Start(void)

DRV_IC_Stop Function

Stops the Input Capture instance for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
void DRV_IC_Stop();
```

Returns

None.

Description

This routine stops the Input Capture driver for the specified driver index, stopping an input capture.

Remarks

None.

Preconditions

DRV_IC_Initialize has been called.

Function

void DRV_IC_Stop(void)

MTCH6301 Driver Library

This topic describes the MTCH6301 Driver Library.

Introduction

This library provides an interface to manage the MTCH6301 Driver module on the Microchip family of microcontrollers in different modes of operation.

Description

The MPLAB Harmony Touch Controller MTCH6301 Driver provides a high-level interface to the touch controller MTCH6301 device. This driver provides application routines to read the touch input data from the touch screen. The MTCH6301 device can notify the availability of touch input data through external interrupt. The MTCH6301 driver allows the application to map a controller pin as an external interrupt pin.

Using the Library

This topic describes the basic architecture of the MTCH6301 Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_mtch6301.h`

The interface to the MTCH6301 Driver library is defined in the `drv_mtch6301.h` header file. Any C language source (`.c`) file that uses the MTCH6301 Driver library should include this header.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the MTCH6301 Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

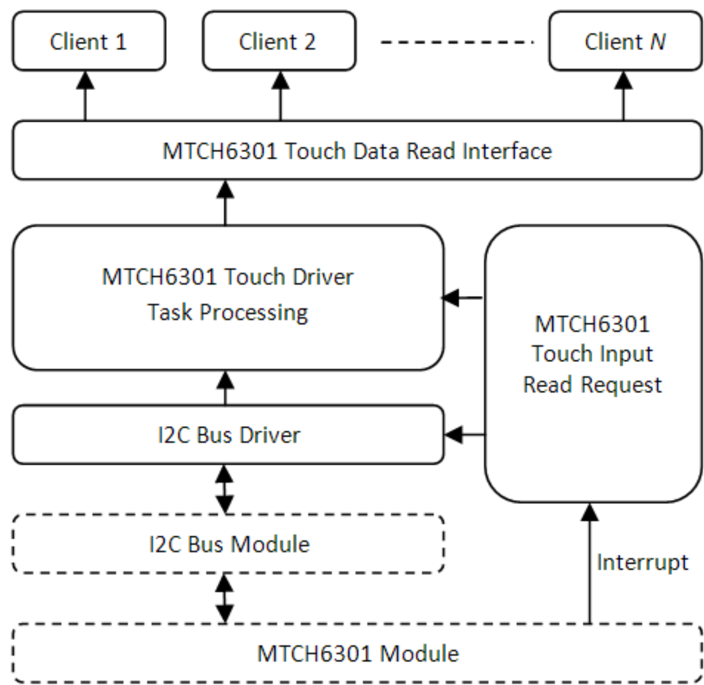
Description

The Touch Controller MTCH6301 Driver has routines to perform the following operations:

- Sending read request
- Reading the touch input data
- Access to the touch input data

The driver initialization routines allow the application to initialize the driver. The driver must be initialized before it can be used by application. Once driver is initialized the driver open routine allows to retrieve the client handle. Once the touch input is available a touch input read request is sent and input data is retrieved in a buffer. The buffer data is then decoded to get the x and y coordinate of the touch screen in the form of the number of pixels.

MTCH6301 Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the MTCH6301 module.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, open, close, task, and status functions.

How the Library Works

The library provides interfaces to support:

- System functions, which provide system module interfaces, device initialization, deinitialization, open, close, task, and status functions.
- Read Request function, which provides Touch input data read request function
- Read Touch Input function, which provides functions retrieving updated Touch input in the form x and y coordinates.

Initializing the Driver

Before the MTCH6301 driver can be opened, it must be configured and initialized. The driver build time configuration is defined by the configuration macros. Refer to the Building the Library section for the location of and more information on the various configuration macros and how these macros should be designed. The driver initialization is configured through the DRV_TOUCH_INIT data structure that is passed to the DRV_TOUCH_MTCH6301_Initialize function. The initialization parameters include the interrupt source, interrupt pin remap configuration and touch screen resolution. The following code shows an example of initializing the MTCH6301 Driver.

Example:

```
/* The following code shows an example of designing the
 *  DRV_TOUCH_INIT data structure. It also shows how an example
 *  usage of the DRV_TOUCH_MTCH6301_Initialize function.
 */

DRV_TOUCH_INIT drvTouchInitData;
SYS_MODULE_OBJ objectHandle;

/* Touch Module Id*/
drvTouchInitData.touchId                     = DRV_TOUCH_INDEX_0;

/* I2C Bus driver open */
drvTouchInitData.drvOpen                     = DRV_I2C_Open;

/* Interrupt Source for Touch */
drvTouchInitData.interruptSource             = INT_SOURCE_EXTERNAL_1;

/* Interrupt Pin function mapping */
drvTouchInitData.interruptPort.inputFunction = INPUT_FUNC_INT1;

/* Pin to be mapped as interrupt pin */
drvTouchInitData.interruptPort.inputPin      = INPUT_PIN_RPE8;

/* Analog pin number */
drvTouchInitData.interruptPort.analogPin     = PORTS_ANALOG_PIN_25;

/* Pin Mode of analog pin */
drvTouchInitData.interruptPort.pinMode       = PORTS_PIN_MODE_DIGITAL;

/* Interrupt pin port */
drvTouchInitData.interruptPort.channel       = PORT_CHANNEL_E;

/* Interrupt pin port maskl */
drvTouchInitData.interruptPort.dataMask      = 0x8;

/* Touch screen orientation */
drvTouchInitData.orientation                 = DISP_ORIENTATION;

/* Touch screen horizontal resolution */
drvTouchInitData.horizontalResolution            = DISP_HOR_RESOLUTION;

/* Touch screen vertical resolution */
drvTouchInitData.verticalResolution          = DISP_VER_RESOLUTION;

/* Driver initialization */
objectHandle = DRV_TOUCH_MTCH6301_Initialize(DRV_TOUCH_INDEX_0,
                                              (SYS_MODULE_INIT*)drvTouchInitData);
    if (SYS_MODULE_OBJ_INVALID == objectHandle)
    {
        // Handle error
    }
```

Opening the Driver

To use the MTCH6301 driver, the application must open the driver. This is done by calling the DRV_TOUCH_MTCH6301_Open function.

If successful, the DRV_TOUCH_MTCH6301_Open function will return a handle to the driver. This handle records the association between the client and the driver instance that was opened. The DRV_TOUCH_MTCH6301_Open function may return DRV_HANDLE_INVALID in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return an invalid handle in other (error) cases as well. The following code shows an example of the driver being opened.

```
DRV_HANDLE  handle;

handle = DRV_TOUCH_MTCH6301_Open( DRV_TOUCH_MTCH6301_INDEX_0,
                                  DRV_IO_INTENT_EXCLUSIVE );

if( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Touch Input Read Request

To read the touch input from the MTCH6301 device, a read request must be registered. This is done by calling the DRV_TOUCH_MTCH6301_ReadRequest. If successful it registers a buffer read request to the I2C command queue. It also adds a input decode command to the MTCH6301 command queue once the I2C returns with touch input data. It can return error if the driver instance object is invalid or the MTCH6301 command queue is full. The read request is to be called from the MTCH6301 ISR. This ISR is triggered once the touch input is available. The following code shows an example of a MTCH6301 read request registration:

```
SYS_MODULE_OBJ      object;   // Returned from DRV_TOUCH_MTCH6301_Initialize

void __ISR(_EXTERNAL_INT_VECTOR, ipl5) _IntHandlerDrvMtch6301(void)
{
    DRV_TOUCH_MTCH6301_ReadRequest ( object );

    // Do other tasks
}
```

Tasks Routine

This routine processes the MTCH6301 commands from the command queue. If the state of the command is initialize or done it returns. If the read request registration is successful the state of command is to decode input. The tasks routine decodes the input and updates the global variables storing the touch input data in form of x and y coordinates. The MTCH6301 driver task routine is to be called from SYS_Tasks. The following code shows an example:

```
SYS_MODULE_OBJ      object;   // Returned from DRV_TOUCH_MTCH6301_Initialize

    void SYS_Tasks( void )
    {
        DRV_TOUCH_MTCH6301_Tasks ( object );

        // Do other tasks
    }
```

Configuring the Library

Macros

	Name	Description
	DRV_MTCH6301_CALIBRATION_DELAY	Define the calibration delay.
	DRV_MTCH6301_CALIBRATION_INSET	Define the calibration inset.
	DRV_MTCH6301_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_MTCH6301_INDEX	MTCH6301 static index selection.
	DRV_MTCH6301_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_MTCH6301_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_MTCH6301_SAMPLE_POINTS	Define the sample points.
	DRV_MTCH6301_TOUCH_DIAMETER	Define the touch diameter.

Description

The configuration of the MTCH6301 Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the MTCH6301 Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the MTCH6301 Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

DRV_MTCH6301_CALIBRATION_DELAY Macro

Define the calibration delay.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_CALIBRATION_DELAY 300
```

Description

MTCH6301 Calibration Delay

This macro enables the delay between calibration touch points.

Remarks

None.

DRV_MTCH6301_CALIBRATION_INSET Macro

Define the calibration inset.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_CALIBRATION_INSET 25
```

Description

MTCH6301 Calibration Inset

This macro define the calibration inset.

Remarks

None.

DRV_MTCH6301_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_CLIENTS_NUMBER 1
```

Description

MTCH6301 Maximum Number of Clients

This definition selected the maximum number of clients that the MTCH6301 driver can support at run time.

Remarks

None.

DRV_MTCH6301_INDEX Macro

MTCH6301 static index selection.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_INDEX DRV_MTCH6301_INDEX_0
```

Description

MTCH6301 Static Index Selection

MTCH6301 static index selection for the driver object reference.

Remarks

This index is required to make a reference to the driver object.

DRV_MTCH6301_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_INSTANCES_NUMBER 1
```

Description

MTCH6301 hardware instance configuration

This macro sets up the maximum number of hardware instances that can be supported.

Remarks

None.

DRV_MTCH6301_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_INTERRUPT_MODE false
```

Description

MTCH6301 Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of MTCH6301 operation is desired
- false - Select if polling mode of MTCH6301 operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

DRV_MTCH6301_SAMPLE_POINTS Macro

Define the sample points.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_SAMPLE_POINTS 4
```

Description

MTCH6301 Sample Points

MTCH6301 sample points

Remarks

None.

DRV_MTCH6301_TOUCH_DIAMETER Macro

Define the touch diameter.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_TOUCH_DIAMETER 10
```

Description

MTCH6301 Touch Diameter

This macro defines the touch diameter

Remarks

None.

Building the Library

This section lists the files that are available in the MTCH6301 Driver Library.

Description

This section list the files that are available in the \src folder of the MTCH6301 Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/touch/mtch6301`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_mtch6301.h	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/drv_mtch6301.c	Basic MTCH6301 Driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The MTCH6301 Driver Library depends on the following modules:

- Interrupt System Service Library
- Ports System Service Library
- Touch System Service Library
- I2C Driver Library

Library Interface

a) System Functions

	Name	Description
	DRV_TOUCH_MTCH6301_Close	Closes an opened instance of the MTCH6301 driver. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Deinitialize	Deinitializes the specified instance of the MTCH6301 driver module. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Initialize	Initializes the MTCH6301 instance for the specified driver index. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Open	Opens the specified MTCH6301 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Status	Provides the current status of the MTCH6301 driver module. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_ReadRequest	Sends a read request to I2C bus driver and adds the read task to queue. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_TouchGetX	Returns the x coordinate of touch input. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_TouchGetY	Returns the y coordinate of touch input. **Implementation:** Dynamic

b) Data Types and Constants

	Name	Description
	_DRV_MTCH6301_CLIENT_OBJECT	MTCH6301 Driver client object maintaining client data.
	DRV_TOUCH_MTCH6301_CLIENT_OBJECT	MTCH6301 Driver client object maintaining client data.
	DRV_TOUCH_MTCH6301_HANDLE	Touch screen controller MTCH6301 driver handle.
	DRV_TOUCH_MTCH6301_MODULE_ID	Number of valid MTCH6301 driver indices.
	DRV_TOUCH_MTCH6301_OBJECT	Defines the data structure maintaining MTCH6301 driver instance object.
	DRV_TOUCH_MTCH6301_TASK_QUEUE	Defines the MTCH6301 Touch Controller driver task data structure.
	DRV_TOUCH_MTCH6301_TASK_STATE	Enumeration defining MTCH6301 touch controller driver task state.
	DRV_TOUCH_MTCH6301_HANDLE_INVALID	Definition of an invalid handle.
	DRV_TOUCH_MTCH6301_I2C_MASTER_READ_ID	MTCH6301 input read, I2C address from where master reads touch input data.
	DRV_TOUCH_MTCH6301_I2C_MASTER_WRITE_ID	MTCH6301 command register write, I2C address where master sends the commands.
	DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE	I2C Frame size for reading MTCH6301 touch input.
	DRV_TOUCH_MTCH6301_INDEX_0	MTCH6301 driver index definitions.

	DRV_TOUCH_MTCH6301_INDEX_1	This is macro DRV_TOUCH_MTCH6301_INDEX_1.
	DRV_TOUCH_MTCH6301_INDEX_COUNT	Number of valid Touch controller MTCH6301 driver indices.

Description

This section describes the API functions of the SPI Driver library.

Refer to each section for a detailed description.

a) System Functions

DRV_TOUCH_MTCH6301_Close Function

Closes an opened instance of the MTCH6301 driver.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
void DRV_TOUCH_MTCH6301_Close(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the MTCH6301 driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_TOUCH_MTCH6301_Open before the caller may use the driver again. This function is thread safe in a RTOS application.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_TOUCH_MTCH6301_Initialize routine must have been called for the specified MTCH6301 driver instance.

DRV_TOUCH_MTCH6301_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TOUCH_MTCH6301_Open

DRV_TOUCH_MTCH6301_Close ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TOUCH_MTCH6301_Close (DRV_HANDLE handle)

DRV_TOUCH_MTCH6301_Deinitialize Function

Deinitializes the specified instance of the MTCH6301 driver module.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
void DRV_TOUCH_MTCH6301_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the MTCH6301 driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_TOUCH_MTCH6301_Status operation. The system has to use DRV_TOUCH_MTCH6301_Status to determine when the module is in the ready state.

Preconditions

Function DRV_TOUCH_MTCH6301_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Parameter: object - Driver object handle, returned from DRV_TOUCH_MTCH6301_Initialize

Example

```
SYS_MODULE_OBJ      object;     //Returned from DRV_TOUCH_MTCH6301_Initialize
SYS_STATUS          status;

DRV_TOUCH_MTCH6301_Deinitialize ( object );

status = DRV_TOUCH_MTCH6301_Status( object );
if( SYS_MODULE_UNINITIALIZED == status )
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Function

void DRV_TOUCH_MTCH6301_Deinitialize (SYS_MODULE_OBJ object)

DRV_TOUCH_MTCH6301_Initialize Function

Initializes the MTCH6301 instance for the specified driver index.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
SYS_MODULE_OBJ DRV_TOUCH_MTCH6301_Initialize(const SYS_MODULE_INDEX index, const
SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the MTCH6301 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the 'init' parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the MTCH6301 module ID. For example, driver instance 0 can be assigned to MTCH63012. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_TOUCH_MTCH6301_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other MTCH6301 routine is called.

This routine should only be called once during system initialization unless DRV_TOUCH_MTCH6301_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
DRV_TOUCH_MTCH6301_INIT        init;
SYS_MODULE_OBJ      objectHandle;

// Populate the MTCH6301 initialization structure
// Touch Module Id
init.touchId                         = DRV_TOUCH_INDEX_0;

// I2C Bus driver open
init.drvOpen                         = DRV_I2C_Open;

// Interrupt Source for Touch
init.interruptSource                 = INT_SOURCE_EXTERNAL_1;

// Interrupt Pin function mapping
init.interruptPort.inputFunction    = INPUT_FUNC_INT1;

// Pin to be mapped as interrupt pin
init.interruptPort.inputPin         = INPUT_PIN_RPE8;

// Analog pin number
init.interruptPort.analogPin        = PORTS_ANALOG_PIN_25;

// Pin Mode of analog pin
init.interruptPort.pinMode          = PORTS_PIN_MODE_DIGITAL;

// Interrupt pin port
init.interruptPort.channel          = PORT_CHANNEL_E;
```

```
// Interrupt pin port maskl
init.interruptPort.dataMask       = 0x8;

// Touch screen orientation
init.orientation                  = DISP_ORIENTATION;

// Touch screen horizontal resolution
init.horizontalResolution         = DISP_HOR_RESOLUTION;

// Touch screen vertical resolution
init.verticalResolution           = DISP_VER_RESOLUTION;


objectHandle = DRV_TOUCH_MTCH6301_Initialize(DRV_TOUCH_INDEX_0,
                                          (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized. Please note this is not the MTCH6301 ID. The hardware MTCH6301 ID is set in the initialization structure. This is the index of the driver index to use.
init	Pointer to a data structure containing any data necessary to initialize the driver. If this pointer is NULL, the driver uses the static initialization override macros for each member of the initialization data structure.

Function

SYS_MODULE_OBJ DRV_TOUCH_MTCH6301_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

DRV_TOUCH_MTCH6301_Open Function

Opens the specified MTCH6301 driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
DRV_HANDLE DRV_TOUCH_MTCH6301_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. An error can occur when the following is true:

- if the number of client objects allocated via DRV_TOUCH_MTCH6301_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid

Description

This routine opens the specified MTCH6301 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options additionally affect the behavior of the DRV_USART_Read() and DRV_USART_Write() functions. If the ioIntent is DRV_IO_INTENT_NONBLOCKING, then these function will not block even if the required amount of data could not be processed. If the ioIntent is DRV_IO_INTENT_BLOCKING, these functions will block until the required amount of data is processed.

If ioIntent is DRV_IO_INTENT_READ, the client will only be read from the driver. If ioIntent is DRV_IO_INTENT_WRITE, the client will only be able to write to the driver. If the ioIntent in DRV_IO_INTENT_READWRITE, the client will be able to do both, read and write.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_TOUCH_MTCH6301_Close routine is called. This routine will NEVER block waiting for hardware. If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

The DRV_TOUCH_MTCH6301_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_TOUCH_MTCH6301_Open( DRV_TOUCH_MTCH6301_INDEX_0,
                                  DRV_IO_INTENT_EXCLUSIVE );

if( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Index of the driver initialized with DRV_TOUCH_MTCH6301_Initialize().
intent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver

Function

```
DRV_HANDLE DRV_TOUCH_MTCH6301_Open ( const SYS_MODULE_INDEX drvIndex,
const                          DRV_IO_INTENT    intent )
```

DRV_TOUCH_MTCH6301_Status Function

Provides the current status of the MTCH6301 driver module.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
SYS_STATUS DRV_TOUCH_MTCH6301_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system-level operation and cannot start another

Description

This function provides the current status of the MTCH6301 driver module.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_MODULE_UNINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

This function can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_TOUCH_MTCH6301_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;   // Returned from DRV_TOUCH_MTCH6301_Initialize
SYS_STATUS          status;

status = DRV_TOUCH_MTCH6301_Status( object );
if( SYS_STATUS_READY != status )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_TOUCH_MTCH6301_Initialize

Function

SYS_STATUS DRV_TOUCH_MTCH6301_Status (SYS_MODULE_OBJ object)

DRV_TOUCH_MTCH6301_Tasks Function

Maintains the driver's state machine and implements its task queue processing.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
void DRV_TOUCH_MTCH6301_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal state machine and implement its command queue processing. It is always called from SYS_Tasks() function. This routine decodes the touch input data available in drvI2CReadFrameData.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks)

Preconditions

The DRV_TOUCH_MTCH6301_Initialize routine must have been called for the specified MTCH6301 driver instance.

Example

```
SYS_MODULE_OBJ      object;   // Returned from DRV_TOUCH_MTCH6301_Initialize

void SYS_Tasks( void )
{
    DRV_TOUCH_MTCH6301_Tasks ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_TOUCH_MTCH6301_Initialize)

Function

void DRV_TOUCH_MTCH6301_Tasks (SYS_MODULE_OBJ object);

DRV_TOUCH_MTCH6301_ReadRequest Function

Sends a read request to I2C bus driver and adds the read task to queue.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
void DRV_TOUCH_MTCH6301_ReadRequest(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to send a touch input read request to the I2C bus driver and adding the input read decode task to the queue. It is always called from MTCH6301 interrupt ISR routine.

Remarks

This function is normally not called directly by an application. It is called by the MTCH6301 ISR routine.

Preconditions

The DRV_TOUCH_MTCH6301_Initialize routine must have been called for the specified MTCH6301 driver instance.

Example

```
SYS_MODULE_OBJ      object;   // Returned from DRV_TOUCH_MTCH6301_Initialize

void __ISR(_EXTERNAL_INT_VECTOR, ipl5) _IntHandlerDrvMtch6301(void)
{
    DRV_TOUCH_MTCH6301_ReadRequest ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_TOUCH_MTCH6301_Initialize)

Function

void DRV_TOUCH_MTCH6301_ReadRequest(SYS_MODULE_OBJ object)

DRV_TOUCH_MTCH6301_TouchGetX Function

Returns the x coordinate of touch input.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
short DRV_TOUCH_MTCH6301_TouchGetX(uint8_t touchNumber);
```

Returns

It returns the x coordinate of the touch input in terms of number of pixels.

Description

It returns the x coordinate in form of number of pixes for a touch input denoted by touchNumber.

Parameters

Parameters	Description
touchNumber	index to the touch input.

Function

short DRV_TOUCH_MTCH6301_TouchGetX(uint8 touchNumber)

DRV_TOUCH_MTCH6301_TouchGetY Function

Returns the y coordinate of touch input.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
short DRV_TOUCH_MTCH6301_TouchGetY(uint8_t touchNumber);
```

Returns

It returns the y coordinate of the touch input in terms of number of pixels.

Description

It returns the y coordinate in form of number of pixes for a touch input denoted by touchNumber.

Parameters

Parameters	Description
touchNumber	index to the touch input.

Function

short DRV_TOUCH_MTCH6301_TouchGetY(uint8 touchNumber)

b) Data Types and Constants

DRV_TOUCH_MTCH6301_CLIENT_OBJECT Structure

MTCH6301 Driver client object maintaining client data.

File

drv_mtch6301.h

C

```
typedef struct _DRV_MTCH6301_CLIENT_OBJECT {
  DRV_TOUCH_MTCH6301_OBJECT* driverObject;
  DRV_IO_INTENT intent;
  struct DRV_TOUCH_MTCH6301_CLIENT_OBJECT* pNext;
} DRV_TOUCH_MTCH6301_CLIENT_OBJECT;
```

Members

Members	Description
DRV_TOUCH_MTCH6301_OBJECT* driverObject;	Driver Object associated with the client
DRV_IO_INTENT intent;	The intent with which the client was opened
struct DRV_TOUCH_MTCH6301_CLIENT_OBJECT* pNext;	Next driver client object

Description

MTCH6301 Driver client object

This defines the object required for the maintenance of the software clients instance. This object exists once per client instance.

Remarks

None.

DRV_TOUCH_MTCH6301_HANDLE Type

Touch screen controller MTCH6301 driver handle.

File

drv_mtch6301.h

C

```
typedef uintptr_t DRV_TOUCH_MTCH6301_HANDLE;
```

Description

MTCH6301 Driver Handle

Touch controller MTCH6301 driver handle is a handle for the driver client object. Each driver with succesful open call will return a new handle to the client object.

Remarks

None.

DRV_TOUCH_MTCH6301_MODULE_ID Enumeration

Number of valid MTCH6301 driver indices.

File

drv_mtch6301.h

C

```
typedef enum {
  MTCH6301_ID_1 = 0,
  MTCH6301_NUMBER_OF_MODULES
} DRV_TOUCH_MTCH6301_MODULE_ID;
```

Description

MTCH6301 Driver Module Index Count

This constant identifies the number of valid MTCH6301 driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific header files defined as part of the peripheral libraries.

DRV_TOUCH_MTCH6301_OBJECT Structure

Defines the data structure maintaining MTCH6301 driver instance object.

File

drv_mtch6301.h

C

```
typedef struct {
  SYS_STATUS status;
  int touchId;
  SYS_MODULE_INDEX drvIndex;
  bool inUse;
  bool isExclusive;
  uint8_t numClients;
  INT_SOURCE interruptSource;
  uint16_t orientation;
  uint16_t horizontalResolution;
  uint16_t verticalResolution;
  DRV_HANDLE (* drvOpen)(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
  int32_t readRequest;
  DRV_TOUCH_MTCH6301_TASK_QUEUE* taskQueue;
  DRV_HANDLE drvI2CHandle;
} DRV_TOUCH_MTCH6301_OBJECT;
```

Members

Members	Description
SYS_STATUS status;	The status of the driver
int touchId;	The peripheral Id associated with the object
SYS_MODULE_INDEX drvIndex;	Save the index of the driver. Important to know this as we are using reference based accessing
bool inUse;	Flag to indicate instance in use
bool isExclusive;	Flag to indicate module used in exclusive access mode
uint8_t numClients;	Number of clients possible with the hardware instance
INT_SOURCE interruptSource;	Touch input interrupt source
uint16_t orientation;	Orientation of the display (given in degrees of 0,90,180,270)
uint16_t horizontalResolution;	Horizontal Resolution of the displayed orientation in Pixels
uint16_t verticalResolution;	Vertical Resolution of the displayed orientaion in Pixels
DRV_HANDLE (* drvOpen)(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);	Callback for I2C Driver Open call
int32_t readRequest;	Touch Input read request counter
DRV_TOUCH_MTCH6301_TASK_QUEUE* taskQueue;	Head of the task queue
DRV_HANDLE drvI2CHandle;	I2C bus driver handle

Description

MTCH6301 Driver Instance Object.

This data structure maintains the MTCH6301 driver instance object. The object exists once per hardware instance.

Remarks

None.

DRV_TOUCH_MTCH6301_TASK_QUEUE Structure

Defines the MTCH6301 Touch Controller driver task data structure.

File

drv_mtch6301.h

C

```
typedef struct {
  bool inUse;
  DRV_TOUCH_MTCH6301_TASK_STATE taskState;
  DRV_I2C_BUFFER_HANDLE drvI2CReadBufferHandle;
  uint8_t drvI2CReadFrameData[DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE];
} DRV_TOUCH_MTCH6301_TASK_QUEUE;
```

Members

Members	Description
bool inUse;	Flag denoting the allocation of task
DRV_TOUCH_MTCH6301_TASK_STATE taskState;	Enum maintaining the task state
DRV_I2C_BUFFER_HANDLE drvI2CReadBufferHandle;	I2C Buffer handle
uint8_t drvI2CReadFrameData[DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE];	Response to Read Touch Input Command • Response = { MTCH6301 Read Address, • Input Data Size, • Touch Id, Pen status, • Touch X coordinate (0 to 6), • Touch X coordinate (7 to 11), • Touch Y coordinate (0 to 6), • Touch Y coordinate (7 to 11) }

Description

MTCH6301 Touch Controller driver task data structure.

This data type defines the data structure maintaing task context in the task queue. The inUse flag denotes the task context allocation for a task. The enum variable taskState maintains the current task state. The I2C buffer handle drvI2CReadBufferHandle maintains the I2C driver buffer handle returned by the I2C driver read request. The byte array variable drvI2CReadFrameData maintains the I2C frame data sent by MTCH6301 after a successful read request.

Remarks

None.

DRV_TOUCH_MTCH6301_TASK_STATE Enumeration

Enumeration defining MTCH6301 touch controller driver task state.

File

drv_mtch6301.h

C

```
typedef enum {
  DRV_TOUCH_MTCH6301_TASK_STATE_INIT = 0,
  DRV_TOUCH_MTCH6301_TASK_STATE_READ_INPUT,
  DRV_TOUCH_MTCH6301_TASK_STATE_DECODE_INPUT,
  DRV_TOUCH_MTCH6301_TASK_STATE_DONE
} DRV_TOUCH_MTCH6301_TASK_STATE;
```

Members

Members	Description
DRV_TOUCH_MTCH6301_TASK_STATE_INIT = 0	Task initialize state
DRV_TOUCH_MTCH6301_TASK_STATE_READ_INPUT	Task read touch input request state
DRV_TOUCH_MTCH6301_TASK_STATE_DECODE_INPUT	Task touch input decode state
DRV_TOUCH_MTCH6301_TASK_STATE_DONE	Task complete state

Description

MTCH6301 Touch Controller Driver Task State

This enumeration defines the MTCH6301 touch controller driver task state. The task state helps to synchronize the operations of initialization the the task, adding the read input task to the task queue once the touch controller notifies the available touch input and a decoding the touch input received.

Remarks

None.

DRV_TOUCH_MTCH6301_HANDLE_INVALID Macro

Definition of an invalid handle.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_HANDLE_INVALID ((DRV_TOUCH_MTCH6301_HANDLE)(-1))
```

Description

MTCH6301 Driver Invalid Handle

This is the definition of an invalid handle. An invalid handle is is returned by DRV_TOUCH_MTCH6301_Open() and DRV_MTCH6301_Close() functions if the request was not successful.

Remarks

None.

DRV_TOUCH_MTCH6301_I2C_MASTER_READ_ID Macro

MTCH6301 input read, I2C address from where master reads touch input data.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_I2C_MASTER_READ_ID 0x4B
```

Description

MTCH6301 Driver Module Master Input Read I2C address

This constant defines the MTCH6301 touch input read I2C address. This address is used as I2C address to read Touch input from MTCH6301 Touch controller.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific data sheets.

DRV_TOUCH_MTCH6301_I2C_MASTER_WRITE_ID Macro

MTCH6301 command register write, I2C address where master sends the commands.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_I2C_MASTER_WRITE_ID 0x4A
```

Description

MTCH6301 Driver Module Master Command Write I2C Address

This constant defines the MTCH6301 command register I2C write address. This address is used as I2C address to write commands into MTCH6301 Touch controller register.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific data sheets.

DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE Macro

I2C Frame size for reading MTCH6301 touch input.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE 7
```

Description

MTCH6301 Driver Module I2C Frame Size

This constant identifies the size of I2C frame required to read from MTCH6301 touch controller. MTCH6301 notifies the availability of input data through interrupt pin.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific data sheets.

DRV_TOUCH_MTCH6301_INDEX_0 Macro

MTCH6301 driver index definitions.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_INDEX_0 0
```

Description

MTCH6301 Driver Module Index Numbers

These constants provide the MTCH6301 driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_MTCH6301_Initialize and DRV_MTCH6301_Open functions to identify the driver instance in use.

DRV_TOUCH_MTCH6301_INDEX_1 Macro

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_INDEX_1 1
```

Description

This is macro DRV_TOUCH_MTCH6301_INDEX_1.

DRV_TOUCH_MTCH6301_INDEX_COUNT Macro

Number of valid Touch controller MTCH6301 driver indices.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_INDEX_COUNT 2
```

Description

MTCH6301 Driver Module Index Count

This constant identifies the number of valid Touch Controller MTCH6301 driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific header files defined as part of the peripheral libraries.

Files

Files

Name	Description
drv_mtch6301.h	Touch controller MTCH6301 Driver interface header file.
drv_mtch6301_config_template.h	MTCH6301 Device Driver configuration template.

Description

This section lists the source and header files used by the SPI Driver Library.

drv_mtch6301.h

Touch controller MTCH6301 Driver interface header file.

Enumerations

	Name	Description
	DRV_TOUCH_MTCH6301_MODULE_ID	Number of valid MTCH6301 driver indices.
	DRV_TOUCH_MTCH6301_TASK_STATE	Enumeration defining MTCH6301 touch controller driver task state.

Functions

	Name	Description
	DRV_TOUCH_MTCH6301_Close	Closes an opened instance of the MTCH6301 driver. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Deinitialize	Deinitializes the specified instance of the MTCH6301 driver module. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Initialize	Initializes the MTCH6301 instance for the specified driver index. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Open	Opens the specified MTCH6301 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_ReadRequest	Sends a read request to I2C bus driver and adds the read task to queue. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Status	Provides the current status of the MTCH6301 driver module. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_TouchGetX	Returns the x coordinate of touch input. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_TouchGetY	Returns the y coordinate of touch input. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_TOUCH_MTCH6301_HANDLE_INVALID	Definition of an invalid handle.

	DRV_TOUCH_MTCH6301_I2C_MASTER_READ_ID	MTCH6301 input read, I2C address from where master reads touch input data.
	DRV_TOUCH_MTCH6301_I2C_MASTER_WRITE_ID	MTCH6301 command register write, I2C address where master sends the commands.
	DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE	I2C Frame size for reading MTCH6301 touch input.
	DRV_TOUCH_MTCH6301_INDEX_0	MTCH6301 driver index definitions.
	DRV_TOUCH_MTCH6301_INDEX_1	This is macro DRV_TOUCH_MTCH6301_INDEX_1.
	DRV_TOUCH_MTCH6301_INDEX_COUNT	Number of valid Touch controller MTCH6301 driver indices.

Structures

	Name	Description
	_DRV_MTCH6301_CLIENT_OBJECT	MTCH6301 Driver client object maintaining client data.
	DRV_TOUCH_MTCH6301_CLIENT_OBJECT	MTCH6301 Driver client object maintaining client data.
	DRV_TOUCH_MTCH6301_OBJECT	Defines the data structure maintaining MTCH6301 driver instance object.
	DRV_TOUCH_MTCH6301_TASK_QUEUE	Defines the MTCH6301 Touch Controller driver task data structure.

Types

	Name	Description
	DRV_TOUCH_MTCH6301_HANDLE	Touch screen controller MTCH6301 driver handle.

Description

Touch Controller MTCH6301 Driver Interface File

This header file describes the macros, data structure and prototypes of the touch controller MTCH6301 driver interface.

File Name

drv_mtch6301.c

drv_mtch6301_config_template.h

MTCH6301 Device Driver configuration template.

Macros

	Name	Description
	DRV_MTCH6301_CALIBRATION_DELAY	Define the calibration delay.
	DRV_MTCH6301_CALIBRATION_INSET	Define the calibration inset.
	DRV_MTCH6301_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_MTCH6301_INDEX	MTCH6301 static index selection.
	DRV_MTCH6301_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_MTCH6301_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_MTCH6301_SAMPLE_POINTS	Define the sample points.
	DRV_MTCH6301_TOUCH_DIAMETER	Define the touch diameter.

Description

MTCH6301 Device Driver Configuration Template

This header file contains the build-time configuration selections for the MTCH6301 device driver. This is the template file which give all possible configurations that can be made. This file should not be included in any project.

File Name

drv_mtch6301_config_template.h

Company

Microchip Technology Inc.

NVM Driver Library

This topic describes the Non-volatile Memory (NVM) Driver Library.

Migrating Applications from v1.03.01 and Earlier Releases of MPLAB Harmony

Provides information on migrating applications from v1.03.01 and earlier releases of MPLAB Harmony to release v1.04 and later.

Description

The NVM Driver Library APIs have changed beginning with the v1.04 release of MPLAB Harmony. Applications that were developed using the earlier version of the MPLAB Harmony NVM Driver (v1.03.01 and earlier) will not build unless the application calls to NVM Driver are updated. While the MHC utility provides an option to continue creating applications using the v1.03.01 and earlier NVM Driver API, it is recommended that existing applications migrate to the latest API to take advantage of the latest features in the NVM Driver. The following sections describe the API changes and other considerations while updating the application for changes in the NVM Driver.

All NVM Driver Demonstration Applications and NVM Driver related documentation have been updated to the latest (new) API. The following sections do not discuss changes in the NVM Driver configuration related code. This code is updated automatically when the project is regenerated using the MHC utility. Only the application related API changes are discussed.

The following table shows the beta API and corresponding v1.04 and Later MPLAB Harmony NVM Driver API.

v1.03.01 and Earlier NVM Driver API	v1.04 and Later NVM Driver API	v1.04 and Later API Notes
DRV_NVM_Initialize	DRV_NVM_Initialize	The init structure now has additional members that allow the NVM media address and geometry to be specified.
DRV_NVM_Deinitialize	DRV_NVM_Deinitialize	No change.
DRV_NVM_Status	DRV_NVM_Status	No change.
DRV_NVM_Open	DRV_NVM_Open	No change.
DRV_NVM_Close	DRV_NVM_Close	No change.
DRV_NVM_Read	DRV_NVM_Read	Parameters have changed: • Returns the command handle associated with the read operation as an output parameter • Data is now read in terms of blocks. The read block size is specified in the NVM Geometry.
DRV_NVM_Write	DRV_NVM_Write	Parameters have changed: • Returns the command handle associated with the write operation as an output parameter • Data is now written in terms of blocks. The write block size is specified in the NVM Geometry.
DRV_NVM_Erase	DRV_NVM_Erase	Parameters have changed: • Returns the command handle associated with the erase operation as an output parameter • NVM Flash is erased in terms of blocks. The erase block size is specified in the NVM Geometry.

DRV_NVM_EraseWrite	DRV_NVM_EraseWrite	Parameters have changed: • Returns the command handle associated with the Erase/Write operation as an output parameter. • Data is now written in terms of blocks. The write block size is specified in the NVM Geometry.
DRV_NVM_BlockEventHandlerSet	DRV_NVM_EventHandlerSet	Function name and parameter type have changed.
DRV_NVM_ClientStatus	Not Available	This API is no longer available.
DRV_NVM_BufferStatus	DRV_NVM_CommandStatus	The DRV_NVM_Read, DRV_NVM_Write, DRV_NVM_Erase, and DRV_NVM_EraseWrite functions now return a command handle associated with the operation. The status of the operation can be checked by passing the command handle to this function.
Not Available	DRV_NVM_GeometryGet	This API gives the following geometrical details of the NVM Flash: • Media Property • Number of Read/Write/Erase regions in the flash device • Number of Blocks and their size in each region of the device
Not Available	DRV_NVM_IsAttached	Returns the physical attach status of the NVM Flash.
Not Available	DRV_NVM_IsWriteProtected	Returns the write protect status of the NVM Flash.
Not Available	DRV_NVM_AddressGet	Returns the NVM Media Start address.

NVM Driver Initialization

DRV_NVM_INIT now takes the following two additional initialization parameters:

- mediaStartAddress - NVM Media Start address. The driver treats this address as the start address for read, write and erase operations.
- nvmMediaGeometry - Indicates the layout of the media in terms of read, write and erase regions.

The following code examples show how the driver initialization was performed with 1.03 APIs and how it is performed with the 1.04 APIs:

Example 1: v1.03 and Earlier Code

```
const DRV_NVM_INIT drvNvmInit =
{
    .moduleInit.sys.powerState = SYS_MODULE_POWER_RUN_FULL,
    .nvmID = NVM_ID_0,
    .interruptSource = INT_SOURCE_FLASH_CONTROL,
};

void SYS_Initialize (void *data)
{
.
.
    // Initialize NVM Driver Layer
    sysObj.drvNvm = DRV_NVM_Initialize(DRV_NVM_INDEX_0, (SYS_MODULE_INIT *)&drvNvmInit);
.
}
```

Example: v1.04 and Later Code

```
/* NVM Geometry structure */
SYS_FS_MEDIA_REGION_GEOMETRY NVMGeometryTable[3] =
```

```
{
    {
        .blockSize = 1,
        .numBlocks = (DRV_NVM_MEDIA_SIZE * 1024),
    },
    {
       .blockSize = DRV_NVM_ROW_SIZE,
       .numBlocks = ((DRV_NVM_MEDIA_SIZE * 1024)/DRV_NVM_ROW_SIZE)
    },
    {
       .blockSize = DRV_NVM_PAGE_SIZE,
       .numBlocks = ((DRV_NVM_MEDIA_SIZE * 1024)/DRV_NVM_PAGE_SIZE)
    }
};

const SYS_FS_MEDIA_GEOMETRY NVMGeometry =
{
    .mediaProperty = SYS_FS_MEDIA_WRITE_IS_BLOCKING,
    .numReadRegions = 1,
    .numWriteRegions = 1,
    .numEraseRegions = 1,
    .geometryTable = (SYS_FS_MEDIA_REGION_GEOMETRY *)&NVMGeometryTable
};

const DRV_NVM_INIT drvNvmInit =
{
    .moduleInit.sys.powerState = SYS_MODULE_POWER_RUN_FULL,
    .nvmID = NVM_ID_0,
    .interruptSource = INT_SOURCE_FLASH_CONTROL,
    .mediaStartAddress = 0x9D010000,
    .nvmMediaGeometry = (SYS_FS_MEDIA_GEOMETRY *)&NVMGeometry
};

void SYS_Initialize (void *data)
{
.
.
    // Initialize NVM Driver Layer
    sysObj.drvNvm = DRV_NVM_Initialize(DRV_NVM_INDEX_0, (SYS_MODULE_INIT *)&drvNvmInit);
.
.
}
```

Addressing in NVM Driver

The v1.03.01 and earlier Read, Write, Erase and EraseWrite APIs took the actual address on which the operation was to be performed. The unit of access was bytes.

In v1.04 the addressing mechanism has been modified. The media start address is set in the DRV_NVM_Initialize. This address is used as the base address for the Read, Write, Erase and EraseWrite APIs. The unit of access is in terms of blocks. The NVM Geometry specifies the media layout in terms of:

- Number of erase, read and write regions
- Block size for erase, read and write operations.
- Number of blocks in erase, read and write regions

For example, in PIC32MZ family devices:

- Read block size = 1 byte
- Write block size = ROW Size = 2048 bytes
- Erase block size = PAGE Size = 16384 bytes

If the size of media is 32 KB then the following table illustrates the address range and number of blocks for the read, write and erase regions:

Region Type	Block Size	Number of blocks	Address range
Read Region	1 Byte	32 KB / Read block size = 32768	0–32767

© 2013-2015 Microchip Technology Inc.

Write Region	2048 Bytes	32 KB / Write block size = 16 blocks	0–15
Erase Region	16384 Bytes	32 KB / Erase block size = 2 blocks	0–1

Erasing Data on NVM Flash

The NVM Geometry indicates the number of erase blocks and the size of a single erase block. The Erase API takes in the erase block start address and the number of blocks to be erased. The following code examples show how to perform the erase operation in v1.03.01 and earlier and how to perform it with v1.04 and later.

Example: v1.03.01 and Earlier Code

```
DRV_HANDLE      myNVMHandle;    // Returned from DRV_NVM_Open
DRV_NVM_BUFFER_HANDLE bufferHandle;

bufferHandle = DRV_NVM_Erase(myNVMHandle, (uint8_t*)NVM_BASE_ADDRESS, DRV_NVM_PAGE_SIZE);
if(DRV_NVM_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Do error handling here
}

// Wait until the buffer completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
while(DRV_NVM_BufferStatus(bufferHandle) != DRV_NVM_BUFFER_COMPLETED);
```

Example: v1.04 and Later Code

```
/* This code example shows how to erase NVM Media data */
DRV_HANDLE nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;
DRV_NVM_COMMAND_STATUS commandStatus;
uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
nBlocks = 1;

DRV_NVM_Erase(nvmHandle, &nvmCommandHandle, blockAddress, nBlocks);
if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the erase request. Handle the error. */
}
// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.

commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Erase completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Erase Failed */
}
```

Writing Data to NVM Flash

The NVM Geometry indicates the number of write blocks and the size of a single write block. The Write API takes in the write block start address and the number of blocks to be written. The following code examples show how the write operation was performed in v1.03.01 and earlier and how to perform it with v1.04 and later APIs:

Example : v1.03.01 and Earlier Code

```
DRV_HANDLE myNVMHandle;    // Returned from DRV_NVM_Open
char       myBuffer[2 * DRV_NVM_ROW_SIZE];

// Destination address should be row aligned.
```

```
char       *destAddress = (char *)NVM_BASE_ADDRESS_TO_WRITE;

unsigned int    count = 2 * MY_BUFFER_SIZE;
DRV_NVM_BUFFER_HANDLE bufferHandle;

bufferHandle  = DRV_NVM_Write(myNVMHandle, destAddress, &myBuffer[total], count);
if(DRV_NVM_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Do error handling here
}

// Wait until the buffer completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
while(DRV_NVM_BufferStatus(bufferHandle) != DRV_NVM_BUFFER_COMPLETED);
```

Example: v1.04 and Later Code

```
/* This code example shows how to write data to NVM Media */
DRV_HANDLE nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;
DRV_NVM_COMMAND_STATUS commandStatus;
uint8_t writeBuf[DRV_NVM_ROW_SIZE];
uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
nBlocks = 1;

DRV_NVM_Write(nvmHandle, &nvmCommandHandle, (uint8_t *)writeBuf, blockAddress, nBlocks);
if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the write request. Handle the error. */
}
// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.

commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Write completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Write Failed */
}
```

Reading Data from NVM Flash

The NVM Geometry indicates the number of read blocks and the size of a single read block. The Read API takes in the read block start address and the number of blocks to be read. The following code examples show how the read operation was performed with v1.03.01 and earlier APIs and how to perform the same with v1.04 and later APIs:

Example: v1.03.01 and Earlier Code

```
DRV_HANDLE      myNVMHandle;    // Returned from DRV_NVM_Open
char            myBuffer[MY_BUFFER_SIZE];
char            *srcAddress = NVM_BASE_ADDRESS_TO_READ_FROM;
unsigned int    count = MY_BUFFER_SIZE;
DRV_NVM_BUFFER_HANDLE bufferHandle;

bufferHandle  = DRV_NVM_Read(myNVMHandle, &myBuffer[total], srcAddress, count);
if(DRV_NVM_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Do error handling here
}

// Wait until the buffer completes. This should not
// be a while loop if a part of cooperative multi-tasking
```

```
// routine. In that case, it should be invoked in task
// state machine.
while(DRV_NVM_BufferStatus(bufferHandle) != DRV_NVM_BUFFER_COMPLETED);
```

Example: v1.04 and Later Code

```
/* This code example shows how to read data from NVM Media */
DRV_HANDLE nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;
DRV_NVM_COMMAND_STATUS commandStatus;
uint8_t readBuf[DRV_NVM_ROW_SIZE];
uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
nBlocks = DRV_NVM_ROW_SIZE;

DRV_NVM_Read(nvmHandle, &nvmCommandHandle, (uint8_t *)readBuf, blockAddress, nBlocks);
if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the read request. Handle the error. */
}
// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.

commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Read completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Read Failed */
}
```

Introduction

The NVM Driver library provides APIs that can be used to interface with the NVM module (controller plus memory) for memory needs.

Description

The NVM Driver provides APIs for block access of the physical media through NVM Driver APIs. As shown in the NVM Driver Abstraction Model, an application or a client can access the physical media using multiple methods, which eventually are facilitated through the NVM Driver.

Memory Devices for PIC Microcontrollers

Depending on the device, there are two primary forms of on-chip memory: Programmable Flash memory and data EEPROM memory. The access mechanism for both of these types are varied.

Flash Program Memory

The Flash program memory is readable, writeable, and erasable during normal operation over the entire operating voltage range.

A read from program memory is executed at one byte/word at a time depending on the width of the data bus.

A write to the program memory is executed in either blocks of specific sizes or a single word depending on the type of processor used.

An erase is performed in blocks. A bulk erase may be performed from user code depending on the type of processor supporting the operation.

Writing or erasing program memory will cease instruction fetches until the operation is complete, restricting memory access, and therefore preventing code execution. This is controlled by an internal programming timer.

There are three processor dependant methods for program memory modification:

- Run-Time Self-Programming (RTSP)
- In-Circuit Serial Programming (ICSP)
- EJTAG programming

This section describes the RTSP techniques.

Using the Library

This topic describes the basic architecture of the NVM Driver Library and provides information and examples on its use.

Description

Interface Header Files: `drv_nvm.h`

The interface to the NVM Driver Library is defined in the `drv_nvm.h` header file. Any C language source (`.c`) file that uses the NVM Driver library should include `drv_nvm.h`.

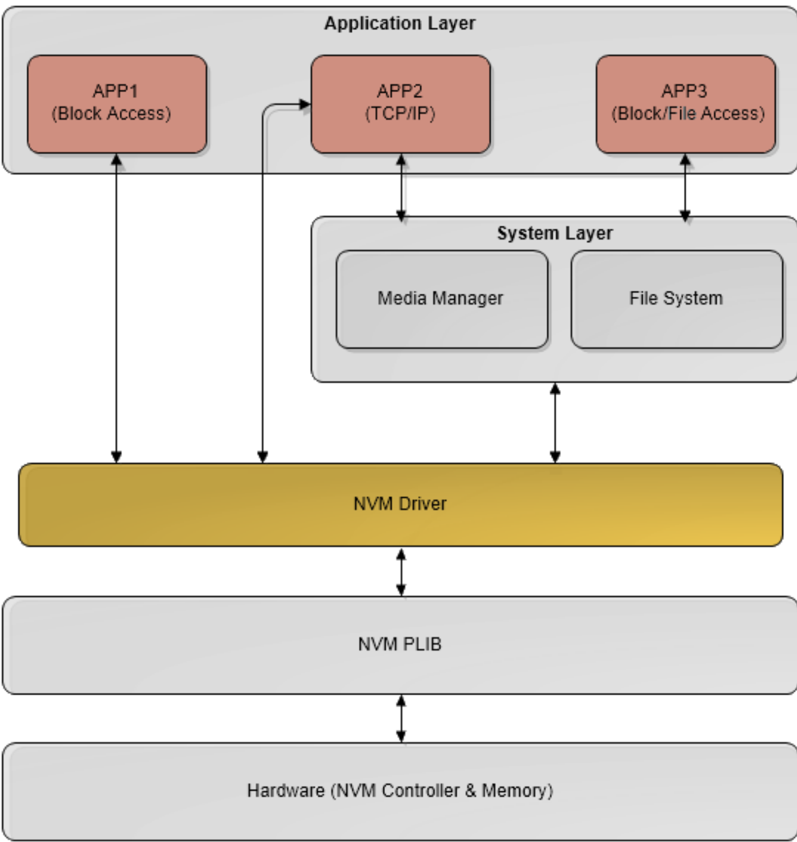
Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the NVM module on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

NVM Driver Abstraction Model



Abstraction Model

As shown in the previous diagram, the NVM Driver sits between the Peripheral Libraries and the application or system layer to facilitate block and file access to the NVM media (currently Flash). The application scenarios show how different layers can be accessed by different applications with certain needs. For example, APP1 can access the

NVM Driver directly to erase, write, or read NVM with direct addressing. APP2, in this case TCP/IP, can bypass the system layer and access the NVM Driver layer if necessary to fulfill its robust data needs. Finally, APP3 accesses the NVM Driver through the File System Layer using block access methods, so the application does not need to keep track of the physical layout of the media.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the NVM module.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Core Functions	Provides open, close, status and other setup functions.
Client Block Transfer Functions	Provides buffered data operation functions available in the core configuration.
Miscellaneous Functions	Provides driver miscellaneous functions related to versions and others.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality
- Media Functionality

Note: Not all modes are available on all devices. Please refer to the specific device data sheet to determine the modes supported for your device.

NVM Driver Initialization

This section provides information for system initialization and reinitialization.

Description

The system performs the initialization and the reinitialization of the device driver with settings that affect only the instance of the device that is being initialized or reinitialized. During system initialization each instance of the NVM module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_NVM_INIT or by using initialization overrides) that are supported by the specific NVM device hardware:

- Device requested power state: One of the system module power states. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- The actual peripheral ID enumerated as the PLIB level module ID (e.g., NVM_ID_0)
- Defining the respective interrupt sources
- NVM Media Start Address
- NVM Media Geometry

The DRV_NVM_Initialize function returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the initialize interface would be used by the other system interfaces, such as DRV_NVM_Deinitialize, DRV_NVM_Status, and DRV_NVM_Tasks.

Note: The system initialization and the reinitialization settings, only affect the instance of the peripheral that is being initialized or reinitialized.

The SYS_MODULE_INDEX is passed to the DRV_NVM_Initialize function to determine which type of memory is selected using: DRV_NVM_INDEX_0 - FLASH

Example:

```
const DRV_NVM_INIT drvNvmInit =
{
    .moduleInit.sys.powerState = SYS_MODULE_POWER_RUN_FULL,
    .nvmID = NVM_ID_0,
    .interruptSource = INT_SOURCE_FLASH_CONTROL,
    .mediaStartAddress = 0x9D010000,
    .nvmMediaGeometry = (SYS_FS_MEDIA_GEOMETRY *)&NVMGeometry
};
void SYS_Initialize (void *data)
{
.
.
.

   // Initialize NVM Driver Layer
     sysObj.drvNvm = DRV_NVM_Initialize(DRV_NVM_INDEX_0, (SYS_MODULE_INIT *)&drvNvmInit);
.
.
.
}
```

Tasks Routine

The system will call DRV_NVM_Tasks, from system task service (in a polled environment) or DRV_NVM_Tasks will be called from the Interrupt Service Routine (ISR) of the NVM.

Client Access Operation

This section provides information for general client operation.

Description

General Client Operation

For the application to start using an instance of the module, it must call the DRV_NVM_Open function. This provides the configuration required to open the NVM instance for operation. If the driver is deinitialized using the function DRV_NVM_Deinitialize, the application must call the DRV_NVM_Open function again to set up the instance of the NVM.

For the various options available for I/O INTENT please refer to **Data Types and Constants** in the Library Interface section.

Example:

```
DRV_HANDLE handle;

handle = DRV_NVM_Open(DRV_NVM_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Client Block Data Operation

This topic provides information on client block data operation.

Description

The NVM Driver provides a block interface to access the NVM media. The interface provides functionality to read, write, erase, and erase-write the NVM media. These interface functions depend on the block sizes and boundaries of the individual devices. The interfaces are responsible for keeping this information transparent from the application.

Erasing Data on the NVM:

The following steps outline the sequence for erasing data on the NVM media:

1. The system should have completed necessary initialization and DRV_NVM_Tasks should either be running in a polled environment, or in an interrupt environment.
2. The driver should have been opened with the necessary intent.
3. Provide the block start address and the number of blocks to be erased and begin the erase process using the DRV_NVM_Erase.
4. The client can check the state of the erase request by invoking the DRV_NVM_CommandStatus and passing the command handle returned by the erase request.
5. The client will be able to close itself by calling the DRV_NVM_Close.

Example:

```
// This code shows how to erase NVM Media data
DRV_HANDLE              nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;

DRV_NVM_COMMAND_STATUS commandStatus;

uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
nBlocks = 1;

DRV_NVM_Erase(nvmHandle, &nvmCommandHandle, blockAddress, nBlocks);

if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the erase request. Handle the error. */
}

// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Erase completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Erase Failed */
}
```

Writing Data to the NVM:

The following steps outline the sequence to be followed for writing data to the NVM Media:

1. The system should have completed necessary initialization and DRV_NVM_Tasks should either be running in a polled environment, or in an interrupt environment.
2. The driver should have been opened with the necessary intent.
3. The client should ensure that blocks of addresses to which write is being performed should be in the erased state.
4. Provide the data to be written, block start address and the number of blocks to be written and begin write using the

DRV_NVM_Write.

5. The client can check the state of the write request by invoking the DRV_NVM_CommandStatus and passing the command handle returned by the write request.
6. The client will be able to close itself by calling the DRV_NVM_Close.

Example:

```
// This code shows how to write data to NVM Media
DRV_HANDLE              nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;

DRV_NVM_COMMAND_STATUS commandStatus;

uint8_t writeBuf[DRV_NVM_ROW_SIZE];
uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
nBlocks = 1;

DRV_NVM_Write(nvmHandle, &nvmCommandHandle, (uint8_t *)writeBuf, blockAddress, nBlocks);

if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the write request. Handle the error. */
}

// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Write completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Write Failed */
}
```

Reading Data from the NVM:

The following steps outline the sequence to be followed for reading data from the NVM Media:

1. The system should have completed necessary initialization and DRV_NVM_Tasks should either be running in a polled environment, or in an interrupt environment.
2. The driver should have been opened with the necessary intent.
3. Provide the target buffer, block start address and the number of blocks to be read and begin reading using the DRV_NVM_Read.
4. The client can check the state of the read request by invoking the DRV_NVM_CommandStatus and passing the command handle returned by the read request.
5. The client will be able to close itself by calling the DRV_NVM_Close.

Example:

```
// This code shows how to read data from NVM Media
DRV_HANDLE              nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;

DRV_NVM_COMMAND_STATUS commandStatus;

uint8_t readBuf[DRV_NVM_ROW_SIZE];
uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
nBlocks = DRV_NVM_ROW_SIZE;
```

```
DRV_NVM_Read(nvmHandle, &nvmCommandHandle, (uint8_t *)readBuf, blockAddress, nBlocks);

if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the read request. Handle the error. */
}

// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.

commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Read completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Read Failed */
}
```

Configuring the Library

Macros

	Name	Description
	DRV_NVM_BUFFER_OBJECT_NUMBER	Selects the maximum number of buffer objects
	DRV_NVM_CLIENTS_NUMBER	Selects the maximum number of clients
	DRV_NVM_INSTANCES_NUMBER	Selects the maximum number of Driver instances that can be supported by the dynamic driver.
	DRV_NVM_INTERRUPT_MODE	Macro specifies operation of the driver to be in the interrupt mode or polled mode
	DRV_NVM_ROW_SIZE	Specifies the NVM Driver Program Row Size in bytes.
	DRV_NVM_UNLOCK_KEY1	Specifies the NVM Driver Program Unlock Key 1
	DRV_NVM_UNLOCK_KEY2	Specifies the NVM Driver Program Unlock Key 2
	DRV_NVM_ERASE_WRITE_ENABLE	Enables support for NVM Driver Erase Write Feature.
	DRV_NVM_PAGE_SIZE	Specifies the NVM Driver Program Page Size in bytes.
	DRV_NVM_DISABLE_ERROR_CHECK	Disables the error checks in the driver.
	DRV_NVM_MEDIA_SIZE	Specifies the NVM Media size.
	DRV_NVM_MEDIA_START_ADDRESS	Specifies the NVM Media start address.
	DRV_NVM_SYS_FS_REGISTER	Register to use with the File system

Description

The configuration of the NVM Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the NVM Driver. Based on the selections made, the NVM Driver may support the selected features. These configuration settings will apply to all instances of the NVM Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

DRV_NVM_BUFFER_OBJECT_NUMBER Macro

Selects the maximum number of buffer objects

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_BUFFER_OBJECT_NUMBER 5
```

Description

NVM Driver maximum number of buffer objects

This definition selects the maximum number of buffer objects. This indirectly also specifies the queue depth. The NVM Driver can queue up DRV_NVM_BUFFER_OBJECT_NUMBER of read/write/erase requests before return a DRV_NVM_BUFFER_HANDLE_INVALID due to the queue being full. Buffer objects are shared by all instances of the driver. Increasing this number increases the RAM requirement of the driver.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_CLIENTS_NUMBER Macro

Selects the maximum number of clients

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_CLIENTS_NUMBER 1
```

Description

NVM maximum number of clients

This definition selects the maximum number of clients that the NVM driver can supported at run time. This constant defines the total number of NVM driver clients that will be available to all instances of the NVM driver.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_INSTANCES_NUMBER Macro

Selects the maximum number of Driver instances that can be supported by the dynamic driver.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_INSTANCES_NUMBER 1
```

Description

NVM Driver instance configuration

This definition selects the maximum number of Driver instances that can be supported by the dynamic driver. In case of this driver, multiple instances of the driver could use the same hardware instance.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_INTERRUPT_MODE Macro

Macro specifies operation of the driver to be in the interrupt mode or polled mode

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_INTERRUPT_MODE true
```

Description

NVM interrupt and polled mode operation control

This macro specifies operation of the driver to be in the interrupt mode or polled mode

- true - Select if interrupt mode of NVM operation is desired
- false - Select if polling mode of NVM operation is desired

Not defining this option to true or false will result in build error.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_ROW_SIZE Macro

Specifies the NVM Driver Program Row Size in bytes.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_ROW_SIZE 512
```

Description

NVM Driver Program Row Size.

This definition specifies the NVM Driver Program Row Size in bytes. This parameter is device specific and should be obtained from the device specific data sheet (Ex: This value is 512 for PIC32MX device variants and 2048 for PIC32MZ device variants). The Program Row Size is the minimum block size that can be programmed in one program operation.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_UNLOCK_KEY1 Macro

Specifies the NVM Driver Program Unlock Key 1

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_UNLOCK_KEY1 0xAA996655
```

Description

NVM Driver Program Unlock Key 1

This definition specifies the NVM Driver Program Unlock Key 1 parameter is device specific and should be obtained from the device specific data sheet.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_UNLOCK_KEY2 Macro

Specifies the NVM Driver Program Unlock Key 2

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_UNLOCK_KEY2 0x556699AA
```

Description

NVM Driver Program Unlock Key 2

This definition specifies the NVM Driver Program Unlock Key 2 parameter is device specific and should be obtained from the device specific data sheet.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_ERASE_WRITE_ENABLE Macro

Enables support for NVM Driver Erase Write Feature.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_ERASE_WRITE_ENABLE
```

Description

NVM Driver Erase Write Feature Enable

Specifying this macro enable row erase write feature. If this macro is specified, the drv_nvm_erasewrite.c file should be added in the project. Support for DRV_NVM_EraseWrite() function then gets enabled.

Remarks

This macro is optional and should be specified only if the DRV_NVM_EraseWrite() function is required.

DRV_NVM_PAGE_SIZE Macro

Specifies the NVM Driver Program Page Size in bytes.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_PAGE_SIZE 4096
```

Description

NVM Driver Program Page Size.

This definition specifies the NVM Driver Program Page Size in bytes. This parameter is device specific and should be obtained from the device specific data sheet(Ex: This value is 4096 for PIC32MX device variants and 16384 for PIC32MZ device variants).

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_DISABLE_ERROR_CHECK Macro

Disables the error checks in the driver.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_DISABLE_ERROR_CHECK
```

Description

NVM Driver Disable Error Checks

Specifying this macro disables the error checks in the driver. Error checks like parameter validation, NULL checks etc, will be disabled in the driver in order to optimize the code space.

Remarks

This macro is optional and should be specified only if code space is a constraint.

DRV_NVM_MEDIA_SIZE Macro

Specifies the NVM Media size.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_MEDIA_SIZE 32
```

Description

NVM Media Size

This definition specifies the NVM Media Size to be used. The size is specified in number of Kilo Bytes. The media size MUST never exceed physical available NVM Memory size. Application code requirements should be kept in mind while defining this parameter.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_MEDIA_START_ADDRESS Macro

Specifies the NVM Media start address.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_MEDIA_START_ADDRESS 0x9D010000
```

Description

NVM Media Start Address

This definition specifies the NVM Media Start address parameter.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_SYS_FS_REGISTER Macro

Register to use with the File system

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_SYS_FS_REGISTER
```

Description

NVM Driver Register with File System

Specifying this macro enables the NVM driver to register its services with the SYS FS.

Remarks

This macro is optional and should be specified only if the NVM driver is to be used with the File System.

Building the Library

This section list the files that are available in the `\src` folder of the NVM driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

Library Interface

a) System Functions

	Name	Description
	DRV_NVM_Initialize	Initializes the NVM instance for the specified driver index
	DRV_NVM_Deinitialize	Deinitializes the specified instance of the NVM driver module
	DRV_NVM_Status	Gets the current status of the NVM driver module.

b) Client Core Functions

	Name	Description
	DRV_NVM_Open	Opens the specified NVM driver instance and returns a handle to it
	DRV_NVM_Close	Closes an opened-instance of the NVM driver
	DRV_NVM_Read	Reads blocks of data from the specified address in memory.
	DRV_NVM_Write	Writes blocks of data starting from the specified address in flash memory.
	DRV_NVM_Erase	Erase the specified number of blocks of the Flash memory.
	DRV_NVM_EraseWrite	Erase and Write blocks of data starting from a specified address in flash memory.
	DRV_NVM_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

c) Client Block Data Functions

	Name	Description
	DRV_NVM_Tasks	Maintains the driver's erase and write state machine and implements its ISR.

d) Status Functions

	Name	Description
	DRV_NVM_AddressGet	Returns the NVM media start address
	DRV_NVM_CommandStatus	Gets the current status of the command.
	DRV_NVM_GeometryGet	Returns the geometry of the device.

e) Miscellaneous Functions

	Name	Description
	DRV_NVM_IsAttached	Returns the physical attach status of the NVM.
	DRV_NVM_IsWriteProtected	Returns the write protect status of the NVM.

f) Data Types and Constants

	Name	Description
	DRV_NVM_INDEX_0	NVM driver index definitions
	DRV_NVM_INIT	Defines the data required to initialize or reinitialize the NVM driver
	DRV_NVM_INDEX_1	This is macro DRV_NVM_INDEX_1.
	DRV_NVM_EVENT	Identifies the possible events that can result from a request.
	DRV_NVM_EVENT_HANDLER	Pointer to a NVM Driver Event handler function
	DRV_NVM_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_NVM_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.
	DRV_NVM_COMMAND_HANDLE_INVALID	This value defines the NVM Driver's Invalid Command Handle.

Description

This section describes the Application Programming Interface (API) functions of the NVM Driver Library.

Refer to each section for a detailed description.

a) System Functions

DRV_NVM_Initialize Function

Initializes the NVM instance for the specified driver index

File

drv_nvm.h

C

```
SYS_MODULE_OBJ DRV_NVM_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const
init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the NVM driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This routine must be called before any other NVM routine is called.

This routine should only be called once during system initialization unless DRV_NVM_Deinitialize is called to deinitialize the driver instance.

This routine will NEVER block for hardware access. If the operation requires time to allow the hardware to reinitialize, it will be reported by the DRV_NVM_Status operation. The system must use DRV_NVM_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this routine.

Preconditions

None.

Example

```
// This code snippet shows an example
// of initializing the NVM Driver.

SYS_MODULE_OBJ  objectHandle;

SYS_FS_MEDIA_REGION_GEOMETRY gNvmGeometryTable[3] =
{
    {
        // Read Region Geometry
        .blockSize = 1,
        .numBlocks = (DRV_NVM_MEDIA_SIZE * 1024),
    },
    {
        // Write Region Geometry
        .blockSize = DRV_NVM_ROW_SIZE,
        .numBlocks = ((DRV_NVM_MEDIA_SIZE * 1024)/DRV_NVM_ROW_SIZE)
    },
    {
        // Erase Region Geometry
        .blockSize = DRV_NVM_PAGE_SIZE,
        .numBlocks = ((DRV_NVM_MEDIA_SIZE * 1024)/DRV_NVM_PAGE_SIZE)
    }
};

const SYS_FS_MEDIA_GEOMETRY gNvmGeometry =
{
    .mediaProperty = SYS_FS_MEDIA_WRITE_IS_BLOCKING,

    // Number of read, write and erase entries in the table
```

```
    .numReadRegions = 1,
    .numWriteRegions = 1,
    .numEraseRegions = 1,
    .geometryTable = &gNvmGeometryTable
};

// FLASH Driver Initialization Data
const DRV_NVM_INIT drvNvmInit =
{
    .moduleInit.sys.powerState = SYS_MODULE_POWER_RUN_FULL,
    .nvmID = NVM_ID_0,
    .interruptSource = INT_SOURCE_FLASH_CONTROL,
    .mediaStartAddress = NVM_BASE_ADDRESS,
    .nvmMediaGeometry = &gNvmGeometry
};

//usage of DRV_NVM_INDEX_0 indicates usage of Flash-related APIs
objectHandle = DRV_NVM_Initialize(DRV_NVM_INDEX_0, (SYS_MODULE_INIT*)&drvNVMInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized also the type of memory used
init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_NVM_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

);

DRV_NVM_Deinitialize Function

Deinitializes the specified instance of the NVM driver module

File

drv_nvm.h

C

```
void DRV_NVM_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the NVM driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

Preconditions

Function DRV_NVM_Initialize should have been called before calling this function.

Parameter: object - Driver object handle, returned from the DRV_NVM_Initialize routine

Example

```
// This code snippet shows an example
// of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_NVM_Initialize
SYS_STATUS          status;

DRV_NVM_Deinitialize(object);

status = DRV_NVM_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Function

void DRV_NVM_Deinitialize

(

SYS_MODULE_OBJ object

);

DRV_NVM_Status Function

Gets the current status of the NVM driver module.

File

drv_nvm.h

C

```
SYS_STATUS DRV_NVM_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations.

SYS_STATUS_UNINITIALIZED - Indicates the driver is not initialized.

Description

This routine provides the current status of the NVM driver module.

Remarks

This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_NVM_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_NVM_Initialize
SYS_STATUS          NVMStatus;

NVMStatus = DRV_NVM_Status(object);
else if (SYS_STATUS_ERROR >= NVMStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_NVM_Initialize routine

Function

SYS_STATUS DRV_NVM_Status

(

SYS_MODULE_OBJ object

);

b) Client Core Functions

DRV_NVM_Open Function

Opens the specified NVM driver instance and returns a handle to it

File

drv_nvm.h

C

```
DRV_HANDLE DRV_NVM_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, DRV_HANDLE_INVALID is returned. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_NVM_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver hardware instance being opened is not initialized or is invalid

Description

This routine opens the specified NVM driver instance and provides a handle. This handle must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_NVM_Close routine is called. This routine will NEVER block waiting for hardware. If the driver has has already been opened, it cannot be opened exclusively.

Preconditions

Function DRV_NVM_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_NVM_Open(DRV_NVM_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_NVM_Open

(

const SYS_MODULE_INDEX index,

const DRV_IO_INTENT ioIntent

);

DRV_NVM_Close Function

Closes an opened-instance of the NVM driver

File

drv_nvm.h

C

```
void DRV_NVM_Close(const DRV_HANDLE handle);
```

Returns

None

Description

This routine closes an opened-instance of the NVM driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_NVM_Open before the caller may use the driver again. Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_NVM_Initialize routine must have been called for the specified NVM driver instance.

DRV_NVM_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_NVM_Open

DRV_NVM_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_NVM_Close

(

const DRV_HANDLE handle

);

DRV_NVM_Read Function

Reads blocks of data from the specified address in memory.

File

drv_nvm.h

C

```
void DRV_NVM_Read(const DRV_HANDLE handle, DRV_NVM_COMMAND_HANDLE * commandHandle, void *
targetBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_NVM_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This routine reads blocks of data from the specified address in memory. This operation is blocking and returns with the required data in the target buffer. If an event handler is registered with the driver the event handler would be invoked from within this function to indicate the status of the operation. This function should not be used to read areas of memory which are queued to be programmed or erased. If required, the program or erase operations should be allowed to complete. The function returns DRV_NVM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the driver handle is invalid
- if the target buffer pointer is NULL
- if the number of blocks to be read is zero or more than the actual number of blocks available
- if a buffer object could not be allocated to the request
- if the client opened the driver in write only mode

Remarks

None.

Preconditions

The DRV_NVM_Initialize routine must have been called for the specified NVM driver instance.

DRV_NVM_Open must have been called with DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE as the ioIntent to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = NVM_BASE_ADDRESS_TO_READ_FROM;
uint32_t    nBlock = 2;
DRV_NVM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myNVMHandle is the handle returned
// by the DRV_NVM_Open function.

DRV_NVM_Read(myNVMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_NVM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
else
{
    // Read Successful
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
targetBuffer	Buffer into which the data read from the NVM Flash instance will be placed
blockStart	Start block address in NVM memory from where the read should begin. It can be any address of the flash.
nBlock	Total number of blocks to be read. Each Read block is of 1 byte.

Function

```
void DRV_NVM_Read
(
const       DRV_HANDLE handle,
      DRV_NVM_COMMAND_HANDLE * commandHandle,
void * targetBuffer,
uint32_t blockStart,
uint32_t nBlock
);
```

DRV_NVM_Write Function

Writes blocks of data starting from the specified address in flash memory.

File

drv_nvm.h

C

```
void DRV_NVM_Write(const DRV_HANDLE handle, DRV_NVM_COMMAND_HANDLE * commandHandle, void *
sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_NVM_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into flash memory. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_NVM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the source buffer pointer is NULL
- if the client opened the driver for read only
- if the number of blocks to be written is either zero or more than the number of blocks actually available
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_NVM_EVENT_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_NVM_EVENT_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

Performing a flash programming operation while executing (fetching) instructions from program Flash memory, the CPU stalls (waits) until the programming operation is finished. The CPU will not execute any instruction, or respond to interrupts, during this time. If any interrupts occur during the programming cycle, they remain pending until the cycle completes. This makes the NVM write operation blocking in nature.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

DRV_NVM_Open() routine must have been called to obtain a valid opened device handle. DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified as a parameter to this routine.

The flash address location which has to be written, must have be erased before using the DRV_NVM_Erase() routine.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = NVM_BASE_ADDRESS_TO_WRITE_TO;
uint32_t    nBlock = 2;
DRV_NVM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myNVMHandle is the handle returned
// by the DRV_NVM_Open function.

// Client registers an event handler with driver
```

```
DRV_NVM_EventHandlerSet(myNVMHandle, APP_NVMEventHandler, (uintptr_t)&myAppObj);

DRV_NVM_Write(myNVMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_NVM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_NVMEventHandler(DRV_NVM_EVENT event,
        DRV_NVM_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_NVM_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_NVM_EVENT_COMMAND_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
sourceBuffer	The source buffer containing data to be programmed into NVM Flash
blockStart	Start block address of NVM Flash where the write should begin. This address should be aligned on a block boundary.
nBlock	Total number of blocks to be written.

Function

void DRV_NVM_Write

(

const DRV_HANDLE handle,

DRV_NVM_COMMAND_HANDLE * commandHandle,

void * sourceBuffer,

uint32_t blockStart,

uint32_t nBlock

);

DRV_NVM_Erase Function

Erase the specified number of blocks of the Flash memory.

File

drv_nvm.h

C

```
void DRV_NVM_Erase(const DRV_HANDLE handle, DRV_NVM_COMMAND_HANDLE * commandHandle, uint32_t
blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It Will be DRV_NVM_COMMAND_HANDLE_INVALID if the request was not queued.

Description

This function schedules a non-blocking erase operation of flash memory. The function returns with a valid erase handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_NVM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the client opened the driver for read only
- if the number of blocks to be erased is either zero or more than the number of blocks actually available
- if the erase queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_NVM_EVENT_COMMAND_COMPLETE event if the erase operation was successful or DRV_NVM_EVENT_COMMAND_ERROR event if the erase operation was not successful.

Remarks

Performing a flash erase operation while executing (fetching) instructions from program Flash memory, the CPU stalls (waits) until the erase operation is finished. The CPU will not execute any instruction, or respond to interrupts, during this time. If any interrupts occur during the programming cycle, they remain pending until the cycle completes. This make the NVM erase operation blocking in nature.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called with DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE to obtain a valid opened device handle.

Example

```
// Destination address should be block aligned.
uint32_t blockStart;
uint32_t nBlock;
DRV_NVM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myNVMHandle is the handle returned
// by the DRV_NVM_Open function.

// Client registers an event handler with driver

DRV_NVM_EventHandlerSet(myNVMHandle, APP_NVMEventHandler, (uintptr_t)&myAppObj);

DRV_NVM_Erase( myNVMHandle, &commandHandle, blockStart, nBlock );

if(DRV_NVM_COMMAND_HANDLE_INVALID == commandHandle)
{
```

```
        // Error handling here
    }

    // Event is received when
    // the buffer queue is processed.

    void APP_NVMEventHandler(DRV_NVM_EVENT event,
            DRV_NVM_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
    {
        // contextHandle points to myAppObj.

        switch(event)
        {
            case DRV_NVM_EVENT_COMMAND_COMPLETE:

                // This means the data was transferred.
                break;

            case DRV_NVM_EVENT_COMMAND_ERROR:

                // Error handling here.
                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
blockStart	Start block address in NVM memory from where the erase should begin. This should be aligned on a DRV_NVM_PAGE_SIZE byte boundary.
nBlock	Total number of blocks to be erased.

Function

void DRV_NVM_Erase

(

const DRV_HANDLE handle,

DRV_NVM_COMMAND_HANDLE * commandHandle,

uint32_t blockStart,

uint32_t nBlock

);

DRV_NVM_EraseWrite Function

Erase and Write blocks of data starting from a specified address in flash memory.

File

drv_nvm.h

C

```
void DRV_NVM_EraseWrite(const DRV_HANDLE handle, DRV_NVM_COMMAND_HANDLE * commandHandle, void *
sourceBuffer, uint32_t writeBlockStart, uint32_t nWriteBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It Will be DRV_NVM_COMMAND_HANDLE_INVALID if the request was not queued.

Description

This function combines the step of erasing a page and then writing the row. The application can use this function if it wants to avoid having to explicitly delete a page in order to update the rows contained in the page.

This function schedules a non-blocking operation to erase and write blocks of data into flash memory. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_NVM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only
- if the buffer size is 0
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_NVM_EVENT_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_NVM_EVENT_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

In order to use this function, the DRV_NVM_ERASE_WRITE_ENABLE must be defined in system_config.h and the drv_nvm_erasewrite.c file must be included in the project.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() must have been called with DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE as a parameter to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = NVM_BASE_ADDRESS_TO_WRITE_TO;
uint32_t    nBlock = 2;
DRV_NVM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myNVMHandle is the handle returned
// by the DRV_NVM_Open function.

// Client registers an event handler with driver

DRV_NVM_EventHandlerSet(myNVMHandle, APP_NVMEventHandler, (uintptr_t)&myAppObj);
```

```
DRV_NVM_EraseWrite(myNVMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_NVM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_NVMEventHandler(DRV_NVM_EVENT event,
        DRV_NVM_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_NVM_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_NVM_EVENT_COMMAND_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle. If NULL, then buffer handle is not returned.
sourceBuffer	The source buffer containing data to be programmed into NVM Flash
writeBlockStart	Start block address of NVM Flash where the write should begin. This address should be aligned on a DRV_NVM_ROW_SIZE byte boundary.
nWriteBlock	Total number of blocks to be written.

Function

void DRV_NVM_EraseWrite

(

const DRV_HANDLE handle,

DRV_NVM_COMMAND_HANDLE * commandHandle,

void * sourceBuffer,

uint32_t writeBlockStart,

uint32_t nWriteBlock

);

DRV_NVM_EventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

File

drv_nvm.h

C

```
void DRV_NVM_EventHandlerSet(const DRV_HANDLE handle, const void * eventHandler, const
uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client calls a write or erase function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any write or erase operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_NVM_COMMAND_HANDLE commandHandle;

// drvNVMHandle is the handle returned
// by the DRV_NVM_Open function.

// Client registers an event handler with driver. This is done once.

DRV_NVM_EventHandlerSet(drvNVMHandle, APP_NVMEventHandler, (uintptr_t)&myAppObj);

DRV_NVM_Read(drvNVMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_NVM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_NVMEventHandler(DRV_NVM_EVENT event,
        DRV_NVM_COMMAND_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;
```

```
    switch(event)
    {
        case DRV_NVM_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_NVM_EVENT_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_NVM_EventHandlerSet

(

const DRV_HANDLE handle,

const void * eventHandler,

const uintptr_t context

);

c) Client Block Data Functions

DRV_NVM_Tasks Function

Maintains the driver's erase and write state machine and implements its ISR.

File

drv_nvm.h

C

```
void DRV_NVM_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal write and erase state machine and implement its ISR for interrupt-driven implementations.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_NVM_Initialize routine must have been called for the specified NVM driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_NVM_Initialize

while (true)
{
    DRV_NVM_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_NVM_Initialize)

Function

void DRV_NVM_Tasks

(

SYS_MODULE_OBJ object

);

d) Status Functions

DRV_NVM_AddressGet Function

Returns the NVM media start address

File

drv_nvm.h

C

```
uintptr_t DRV_NVM_AddressGet(const DRV_HANDLE handle);
```

Returns

Start address of the NVM Media if the handle is valid otherwise NULL.

Description

This function returns the NVM Media start address.

Remarks

None.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
uintptr_t startAddress;
startAddress = DRV_NVM_AddressGet(drvNVMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

uintptr_t DRV_NVM_AddressGet

(

const DRV_HANDLE handle

);

DRV_NVM_CommandStatus Function

Gets the current status of the command.

File

drv_nvm.h

C

```
DRV_NVM_COMMAND_STATUS DRV_NVM_CommandStatus(const DRV_HANDLE handle, const
DRV_NVM_COMMAND_HANDLE commandHandle);
```

Returns

A DRV_NVM_COMMAND_STATUS value describing the current status of the command. Returns DRV_NVM_COMMAND_HANDLE_INVALID if the client handle or the command handle is not valid.

Description

This routine gets the current status of the command. The application must use this routine where the status of a scheduled command needs to polled on. The function may return DRV_NVM_COMMAND_HANDLE_INVALID in a case where the command handle has expired. A command handle expires when the internal buffer object is re-assigned to another erase or write request. It is recommended that this function be called regularly in order to track the command status correctly.

The application can alternatively register an event handler to receive write or erase operation completion events.

Remarks

This routine will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_NVM_Initialize() routine must have been called.

The DRV_NVM_Open() must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE                  handle;         // Returned from DRV_NVM_Open
DRV_NVM_COMMAND_HANDLE      commandHandle;
DRV_NVM_COMMAND_STATUS      status;

status = DRV_NVM_CommandStatus(handle, commandHandle);
if(status == DRV_NVM_COMMAND_COMPLETED)
{
    // Operation Done
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_NVM_COMMAND_STATUS DRV_NVM_CommandStatus

(

const DRV_HANDLE handle,

const DRV_NVM_COMMAND_HANDLE commandHandle

);

DRV_NVM_GeometryGet Function

Returns the geometry of the device.

File

drv_nvm.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_NVM_GeometryGet(const DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Pointer to structure which holds the media geometry information.

Description

This API gives the following geometrical details of the NVM Flash:

- Media Property
- Number of Read/Write/Erase regions in the flash device
- Number of Blocks and their size in each region of the device

Remarks

None.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
SYS_FS_MEDIA_GEOMETRY * nvmFlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

nvmFlashGeometry = DRV_NVM_GeometryGet(nvmOpenHandle1);

readBlockSize  = nvmFlashGeometry->geometryTable->blockSize;
nReadBlocks = nvmFlashGeometry->geometryTable->numBlocks;
nReadRegions = nvmFlashGeometry->numReadRegions;

writeBlockSize  = (nvmFlashGeometry->geometryTable +1)->blockSize;
eraseBlockSize  = (nvmFlashGeometry->geometryTable +2)->blockSize;

totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY * DRV_NVM_GeometryGet

(

const DRV_HANDLE handle

);

e) Miscellaneous Functions

DRV_NVM_IsAttached Function

Returns the physical attach status of the NVM.

File

drv_nvm.h

C

```
bool DRV_NVM_IsAttached(const DRV_HANDLE handle);
```

Returns

Returns false if the handle is invalid otherwise returns true.

Description

This function returns the physical attach status of the NVM.

Remarks

None.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// The NVM media is always attached and so the below
// always returns true.

bool isNVMAttached;
isNVMAttached = DRV_NVM_isAttached(drvNVMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_NVM_IsAttached

(

const DRV_HANDLE handle

);

DRV_NVM_IsWriteProtected Function

Returns the write protect status of the NVM.

File

drv_nvm.h

C

```
bool DRV_NVM_IsWriteProtected(const DRV_HANDLE handle);
```

Returns

Always returns false.

Description

This function returns the physical attach status of the NVM. This function always returns false.

Remarks

None.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// The NVM media is treated as always writeable.
bool isWriteProtected;
isWriteProtected = DRV_NVM_IsWriteProtected(drvNVMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_NVM_IsWriteProtected

(

const DRV_HANDLE handle

);

f) Data Types and Constants

DRV_NVM_INDEX_0 Macro

NVM driver index definitions

File

drv_nvm.h

C

```
#define DRV_NVM_INDEX_0 0
```

Description

Driver NVM Module Index reference

These constants provide NVM driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_NVM_Initialize and DRV_NVM_Open routines to identify the driver instance in use.

DRV_NVM_INIT Structure

Defines the data required to initialize or reinitialize the NVM driver

File

drv_nvm.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  NVM_MODULE_ID nvmID;
  INT_SOURCE interruptSource;
  uint32_t mediaStartAddress;
  const SYS_FS_MEDIA_GEOMETRY * nvmMediaGeometry;
} DRV_NVM_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
NVM_MODULE_ID nvmID;	Identifies NVM hardware module (PLIB-level) ID
INT_SOURCE interruptSource;	Interrupt Source for Write Interrupt
uint32_t mediaStartAddress;	NVM Media start address. The driver treats this address as • block 0 address for read, write and erase operations.
const SYS_FS_MEDIA_GEOMETRY * nvmMediaGeometry;	NVM Media geometry object.

Description

NVM Driver Initialization Data

This data type defines the data required to initialize or reinitialize the NVM driver.

Remarks

Not all initialization features are available for all devices. Please refer to the specific device data sheet to determine availability.

DRV_NVM_INDEX_1 Macro

File

drv_nvm.h

C

```
#define DRV_NVM_INDEX_1 1
```

Description

This is macro DRV_NVM_INDEX_1.

DRV_NVM_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_nvm.h

C

```
typedef enum {
  DRV_NVM_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_NVM_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR
} DRV_NVM_EVENT;
```

Members

Members	Description
DRV_NVM_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE	Operation has been completed successfully.
DRV_NVM_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR	There was an error during the operation

Description

NVM Driver Events

This enumeration identifies the possible events that can result from a Write or Erase request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_NVM_EventHandlerSet function when a request is completed.

DRV_NVM_EVENT_HANDLER Type

Pointer to a NVM Driver Event handler function

File

drv_nvm.h

C

```
typedef SYS_FS_MEDIA_EVENT_HANDLER DRV_NVM_EVENT_HANDLER;
```

Returns

None.

Description

NVM Driver Event Handler Function Pointer

This data type defines the required function signature for the NVM event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_NVM_EVENT_COMMAND_COMPLETE, it means that the write or a erase operation was completed successfully.

If the event is DRV_NVM_EVENT_COMMAND_ERROR, it means that the scheduled operation was not completed successfully.

The context parameter contains the handle to the client context, provided at the time the event handling function was registered using the DRV_NVM_EventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations within this function.

Example

```
void APP_MyNvmEventHandler
(
    DRV_NVM_EVENT event,
    DRV_NVM_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_NVM_EVENT_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_NVM_EVENT_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write/Erase requests
context	Value identifying the context of the application that registered the event handling function

DRV_NVM_COMMAND_HANDLE Type

Handle identifying commands queued in the driver.

File

drv_nvm.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_NVM_COMMAND_HANDLE;
```

Description

NVM Driver command handle.

A command handle is returned by a call to the Read, Write or Erase functions. This handle allows the application to track the completion of the operation. This command handle is also returned to the client along with the event that has occurred with respect to the command. This allows the application to connect the event to a specific command in case where multiple commands are queued.

The command handle associated with the command request expires when the client has been notified of the completion of the command (after event handler function that notifies the client returns) or after the command has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_NVM_COMMAND_STATUS Enumeration

Specifies the status of the command for the read, write and erase operations.

File

drv_nvm.h

C

```
typedef enum {
  DRV_NVM_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED,
  DRV_NVM_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED,
  DRV_NVM_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS,
  DRV_NVM_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN
} DRV_NVM_COMMAND_STATUS;
```

Members

Members	Description
DRV_NVM_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED	Done OK and ready
DRV_NVM_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED	Scheduled but not started
DRV_NVM_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_NVM_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN	Unknown Command

Description

NVM Driver Command Status

NVM Driver command Status

This type specifies the status of the command for the read, write and erase operations.

Remarks

None.

DRV_NVM_COMMAND_HANDLE_INVALID Macro

This value defines the NVM Driver's Invalid Command Handle.

File

drv_nvm.h

C

```
#define DRV_NVM_COMMAND_HANDLE_INVALID SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID
```

Description

NVM Driver Invalid Command Handle.

This value defines the NVM Driver Invalid Command Handle. This value is returned by read/write/erase routines when the command request was not accepted.

Remarks

None.

Files

Files

Name	Description
drv_nvm.h	NVM Driver Interface Definition
drv_nvm_config_template.h	NVM driver configuration definitions.

Description

This section lists the source and header files used by the NVM Driver Library.

drv_nvm.h

NVM Driver Interface Definition

Enumerations

	Name	Description
	DRV_NVM_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.
	DRV_NVM_EVENT	Identifies the possible events that can result from a request.

Functions

	Name	Description
	DRV_NVM_AddressGet	Returns the NVM media start address
	DRV_NVM_Close	Closes an opened-instance of the NVM driver
	DRV_NVM_CommandStatus	Gets the current status of the command.
	DRV_NVM_Deinitialize	Deinitializes the specified instance of the NVM driver module
	DRV_NVM_Erase	Erase the specified number of blocks of the Flash memory.
	DRV_NVM_EraseWrite	Erase and Write blocks of data starting from a specified address in flash memory.
	DRV_NVM_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.
	DRV_NVM_GeometryGet	Returns the geometry of the device.
	DRV_NVM_Initialize	Initializes the NVM instance for the specified driver index
	DRV_NVM_IsAttached	Returns the physical attach status of the NVM.
	DRV_NVM_IsWriteProtected	Returns the write protect status of the NVM.
	DRV_NVM_Open	Opens the specified NVM driver instance and returns a handle to it
	DRV_NVM_Read	Reads blocks of data from the specified address in memory.
	DRV_NVM_Status	Gets the current status of the NVM driver module.
	DRV_NVM_Tasks	Maintains the driver's erase and write state machine and implements its ISR.
	DRV_NVM_Write	Writes blocks of data starting from the specified address in flash memory.

Macros

	Name	Description
	DRV_NVM_COMMAND_HANDLE_INVALID	This value defines the NVM Driver's Invalid Command Handle.
	DRV_NVM_INDEX_0	NVM driver index definitions
	DRV_NVM_INDEX_1	This is macro DRV_NVM_INDEX_1.

Structures

	Name	Description
	DRV_NVM_INIT	Defines the data required to initialize or reinitialize the NVM driver

Types

	Name	Description
	DRV_NVM_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_NVM_EVENT_HANDLER	Pointer to a NVM Driver Event handler function

Description

NVM Driver Interface Definition

The NVM driver provides a simple interface to manage the Non Volatile Flash Memory on Microchip microcontrollers. This file defines the interface definition for the NVM driver.

File Name

drv_nvm.h

Company

Microchip Technology Inc.

drv_nvm_config_template.h

NVM driver configuration definitions.

Macros

	Name	Description
	DRV_NVM_BUFFER_OBJECT_NUMBER	Selects the maximum number of buffer objects
	DRV_NVM_CLIENTS_NUMBER	Selects the maximum number of clients
	DRV_NVM_DISABLE_ERROR_CHECK	Disables the error checks in the driver.
	DRV_NVM_ERASE_WRITE_ENABLE	Enables support for NVM Driver Erase Write Feature.
	DRV_NVM_INSTANCES_NUMBER	Selects the maximum number of Driver instances that can be supported by the dynamic driver.
	DRV_NVM_INTERRUPT_MODE	Macro specifies operation of the driver to be in the interrupt mode or polled mode
	DRV_NVM_MEDIA_SIZE	Specifies the NVM Media size.
	DRV_NVM_MEDIA_START_ADDRESS	Specifies the NVM Media start address.
	DRV_NVM_PAGE_SIZE	Specifies the NVM Driver Program Page Size in bytes.
	DRV_NVM_ROW_SIZE	Specifies the NVM Driver Program Row Size in bytes.
	DRV_NVM_SYS_FS_REGISTER	Register to use with the File system
	DRV_NVM_UNLOCK_KEY1	Specifies the NVM Driver Program Unlock Key 1
	DRV_NVM_UNLOCK_KEY2	Specifies the NVM Driver Program Unlock Key 2

Description

NVM Driver Configuration Template Header file.

This template file describes all the mandatory and optional configuration macros that are needed for building the NVM driver. Do not include this file in source code.

File Name

drv_nvm_config_template.h

Company

Microchip Technology Inc.

Output Compare Driver Library

This topic describes the Output Compare Driver Library.

Introduction

The Output Compare Static Driver provides a high-level interface to manage the Output Compare module on the Microchip family of microcontrollers.

Description

Through the MHC, this driver provides APIs for the following:

- Initializing the module
- Enabling/Disabling of the output compare
- Starting/Stopping of the output compare
- Fault checking

Library Interface

Functions

	Name	Description
	DRV_OC_Disable	Disables the Output Compare instance for the specified driver index. **Implementation:** Static
	DRV_OC_Enable	Enables the Output Compare for the specified driver index. **Implementation:** Static
	DRV_OC_FaultHasOccurred	Checks if a Fault has occurred for the specified driver index. **Implementation:** Static
	DRV_OC_Initialize	Initializes the Comparator instance for the specified driver index. **Implementation:** Static
	DRV_OC_Start	Starts the Comparator instance for the specified driver index. **Implementation:** Static
	DRV_OC_Stop	Stops the Output Compare instance for the specified driver index. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the Output Compare Driver Library.

Functions

DRV_OC_Disable Function

Disables the Output Compare instance for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
void DRV_OC_Disable();
```

Returns

None.

Description

This routine disables the Output Compare for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

None.

Preconditions

DRV_OC_Initialize has been called.

Function

void DRV_OC_Disable(void)

DRV_OC_Enable Function

Enables the Output Compare for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
void DRV_OC_Enable();
```

Returns

None.

Description

This routine enables the Output Compare for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

None.

Preconditions

DRV_OC_Initialize has been called.

Function

void DRV_OC_Enable(void)

DRV_OC_FaultHasOccurred Function

Checks if a Fault has occurred for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
bool DRV_OC_FaultHasOccurred();
```

Returns

Boolean

- 1 - A Fault has occurred
- 0 - A Fault has not occurred

Description

This routine checks whether or not a Fault has occurred for the specified driver index. The initialization routine is specified by the MHC parameters.

Remarks

None.

Preconditions

DRV_OC_Initialize has been called.

Function

bool DRV_OC_FaultHasOccurred(void)

DRV_OC_Initialize Function

Initializes the Comparator instance for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
void DRV_OC_Initialize();
```

Returns

None.

Description

This routine initializes the Output Compare driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters. The driver instance index is independent of the Output Compare module ID. For example, driver instance 0 can be assigned to Output Compare 1.

Remarks

This routine must be called before any other Comparator routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_OC_Initialize(void)

DRV_OC_Start Function

Starts the Comparator instance for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
void DRV_OC_Start();
```

Returns

None.

Description

This routine starts the Output Compare for the specified driver instance.

Remarks

None.

Preconditions

DRV_OC_Initialize has been called.

Function

void DRV_OC_Start(void)

DRV_OC_Stop Function

Stops the Output Compare instance for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
void DRV_OC_Stop();
```

Returns

None.

Description

This routine stops the Output Compare for the specified driver instance.

Remarks

None.

Preconditions

DRV_OC_Initialize has been called.

Function

void DRV_OC_Stop(void)

Parallel Master Port (PMP) Driver Library

This topic describes the Parallel Master Port Driver Library.

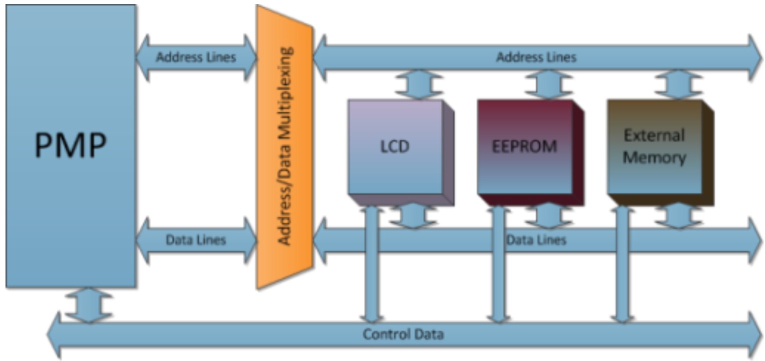
Introduction

This library provides an interface to manage the Parallel Master Port (PMP) module on Microchip family of microcontrollers in different modes of operation.

Description

The Parallel Master Port (PMP) is a parallel 8-bit/16-bit I/O module specifically designed to communicate with a wide variety of parallel devices such as communications peripherals, LCDs, external memory devices and microcontrollers. Because the interfaces to parallel peripherals vary significantly, the PMP module is highly configurable.

The following figure shows a generic block diagram, which illustrates the ways the PMP module can be used:



The PMP module can be used in different modes. Master and Slave are the two modes that can have additional sub-modes, depending on the different microcontroller families.

Master Mode: In Master mode, the PMP module can provide a 8-bit or 16-bit data bus, up to 16 bits of address, and all of the necessary control signals to operate a variety of external parallel devices such as memory devices, peripherals and slave microcontrollers. The PMP master modes provide a simple interface for reading and writing data, but not executing program instructions from external devices, such as SRAM or Flash memories.

Slave Mode: Slave mode only supports 8-bit data and the module control pins are automatically dedicated when this mode is selected.

Using the Library

This topic describes the basic architecture of the PMP Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_pmp.h`

The interface to the PMP Driver library is defined in the `drv_pmp.h` header file. This file is included by the `drv.h` file. Any C language source (`.c`) file that uses the PMP Driver Library should include `drv.h`.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

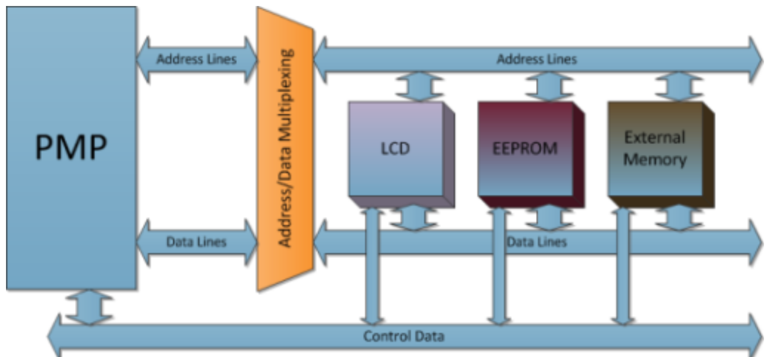
This library provides a low-level abstraction of the Parallel Master Port (PMP) module on Microchip's microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

Hardware Abstraction Model Description

Depending on the device,the PMP module provides interface routines to interact with external peripherals such as LCD, EEPROM, Flash memory, etc., as shown in the following diagram. The diagram shows the PMP module acting as a master. The PMP module can be easily configured to act as a slave. The address and data lines can be multiplexed to suit the application. The address and data buffers are up to 2-byte (16-bit) buffers for data transmitted or received by the parallel interface to the PMP bus over the data and address lines synchronized with control logic including the read and write strobe.

The desired timing wait states to suit different peripheral timings can also be programmed using the PMP module.



PMP Hardware Abstraction Model Diagram

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the PMP module.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks, and status functions.

Client Interaction Functions	Provides open, close, client status and client mode configuration functions.
Client Transfer Functions	Provides interface for data transfer in master and slave mode.
Miscellaneous	Provides driver miscellaneous functions, version identification functions, etc.

How the Library Works

This section describes how the PMP Driver Library operates.

Description

Before the driver is ready for use, its should be configured (compile time configuration). Refer to the Configuring the Library section for more details on how to configure the driver.

There are few run-time configuration items that are done during initialization of the driver instance, and a few that are client-specific and are done using dedicated functions.

To use the PMP Driver, initialization and client functions should be invoked in a specific sequence to ensure correct operation.

The following is the sequence in which various routines should be called:

1. Call DRV_PMP_Initialize to initialize the PMP Driver. Note that this may be performed by the MPLAB Harmony system module. The DRV_PMP_Status function may be used to check the status of the initialization.
2. Once initialization for a particular driver instance is done, the client wanting to use the driver can open it using DRV_PMP_Open.
3. The DRV_PMP_ModeConfig function should now be called, which will configure the driver for the exact mode of operation required by that client.
4. After configuring the mode, DRV_PMP_Write and/or DRV_PMP_Read can be called by the user application to Write/Read using the PMP module. Calling these functions does not start the PMP transfer immediately in non-interrupt mode. Instead, all of these transfer tasks are queued in an internal queue. Actual transfer starts only when the PMP Task function is called by the system/user. In interrupt mode, although transfer tasks are queued, the actual transfer starts immediately.
5. PMP Write and Read functions return an ID of that particular transfer, which should be saved by user to get the status of that transfer later.
6. The system will either call DRV_PMP_Tasks from the System Task Service (in a polled environment), or it will be called from the ISR of the PMP.
7. At any time status of the transfer can be obtained by using DRV_PMP_TransferStatus.

Note: Not all modes are available on all devices. Please refer to the specific device data sheet to determine the supported modes.

System Initialization

This section describes initialization and reinitialization features.

Description

Initialization and Reinitialization

The system performs the initialization and the reinitialization of the device driver with settings that affect only the instance of the device that is being initialized or reinitialized. During system initialization each instance of the PMP device will be initialized with the following configuration settings:

Initialization Member	Description
moduleInit	System module initialization of the power state.
pmpId	PMP hardware module ID (peripheral library-level ID).
stopInIdle	Decide whether or not the module should be stopped in Idle mode.
muxMode	To select one of the different multiplexing modes possible for PMP module.
inputBuffer	Select the type of Input Buffer (TTL or Schmitt Trigger).
polarity	Select polarity of different PMP pins.
ports	Set the pins the user wants to use as port or PMP pins.

The DRV_PMP_Initialize function returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the initialize interface would be used by the other system interfaces, such as DRV_PMP_Reinitialize, DRV_PMP_Deinitialize, DRV_PMP_Status, and DRV_PMP_Tasks.

Example for PMP Initialization Through the DRV_PMP_INIT Structure

```
DRV_PMP_INIT      init;
SYS_MODULE_OBJ    object;
SYS_STATUS        pmpStatus;

// populate the PMP init configuration structure
    init.inputBuffer = PMP_INPUT_BUFFER_TTL;
    init.polarity.addressLatchPolarity = PMP_POLARITY_ACTIVE_HIGH;
    init.polarity.rwStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
    init.polarity.writeEnableStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
    init.polarity.chipselect1Polarity = PMP_POLARITY_ACTIVE_HIGH;
    init.polarity.chipselect2Polarity = PMP_POLARITY_ACTIVE_LOW;
    init.ports.addressPortsMask = PMP_PMA0_PORT | PMP_PMA1_PORT | PMP_PMA2_TO_PMA13_PORTS | PMP_PMA14_PORT;
    init.ports.readWriteStrobe = PORT_ENABLE;
    init.ports.writeEnableStrobe = PORT_ENABLE;
    init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
    init.pmpID              = PMP_ID_0;
    init.stopInIdle         = false;
    init.muxMode            = PMP_MUX_NONE;

object = DRV_PMP_Initialize (DRV_PMP_INDEX_0, (SYS_MODULE_INIT *)&init);

pmpStatus = DRV_PMP_Status(object);

if ( SYS_STATUS_READY != pmpStatus)
{
    // Handle error
}
```

Deinitialization

Once the initialize operation has been called, the deinitialize operation must be called before the initialize operation can be called again. This routine may block if the driver is running in an OS environment that supports blocking operations and the driver requires system resources access. However, the function will never block for hardware PMP access. If the operation requires time to allow the hardware to complete, which will be reported by DRV_PMP_Status.

Status

PMP status is available to query the module state before, during and after initialization, deinitialization, and reinitialization.

Tasks Routine

The DRV_PMP_Tasks function will see the queue status and perform the task of transferring the data accordingly. In the Blocking mode when interrupts are disabled, it will finish one of the tasks completely (that means emptying one space in queue), and then return back. Whereas in Non-Blocking mode, it will return back just after starting one word (8-bit or 16-bit) of transfer (may not be emptying one space in the queue, as that task may not be completely finished).

The DRV_PMP_Tasks function can be called in two ways:

- By the system task service in a polled environment
- By the ISR of the PMP in an interrupt-based system

Example: Polling

```
int main( void )
{
    SYS_MODULE_OBJ object;
    object = DRV_PMP_Initialize( DRV_PMP_INDEX_0, (SYS_MODULE_INIT *) &initConf );

    if( SYS_STATUS_READY != DRV_PMP_Status( object ) )
        return 0;

    while (1)
    {
        DRV_PMP_Tasks (object);
    }
}
```

Example: Interrupt

```
int main( void )
{
    SYS_MODULE_OBJ object;
    object = DRV_PMP_Initialize( DRV_PMP_INDEX_0, (SYS_MODULE_INIT *) &initConf );

    if( SYS_STATUS_READY != DRV_PMP_Status( object ) )
        return 0;

    while (1);
}

/* Sample interrupt routine not specific to any device family */
void ISR PMPInterrupt(void)
{
    //Call the PMP Tasks routine
    DRV_PMP_Tasks(object);
}
```

Note: A PMP transfer in Blocking mode in an interrupt environment is not supported.

Transfer Operation

This section describes transfer operation.

Description

Once the PMP Driver is open and configured for a client, it is set to start Reading/Writing through DRV_PMP_Read and DRV_PMP_Write. However, these functions will not directly start reading or writing. These will just put the relevant information in a queue in non-interrupt mode and return an ID that can be used later for checking the transfer status. In Interrupt mode, the Read/Write functions will trigger the transfer immediately after storing the

transfer information in the queue.

The user must use a buffer pointing to character for data values.

The repeatCount parameter allows the user to repeatedly write the same nBytes of data into the slave devices.

Example:

```
unsigned char myReadBuffer[300], myWriteBuffer[100]; // has to be 'char' arrays
uint32_t deviceAddress, nBytes, repeatCount, i;
uint32_t writeID, readID;
DRV_HANDLE handle;

//initialize, open and configure the driver/client
/* ... */

deviceAddress = 0x0206;
nBytes = 100;
repeatCount = 0x01;
for (i=0; i<nBytes; i++)
{
   myWriteBuffer[i]=i*5+7;
}

/* it will write 100 bytes of data in the location starting from 0x0206 and then it will repeat
 writing the same set of data in next 100 location starting from 0x206+100 for 8 bit data mode
 and 50 location starting from 0x206+50 for 16 bit data mode. */
writeID = DRV_PMP_Write ( handle, deviceAddress, &myWriteBuffer[0], nBytes, repeatCount);

// it will read 300 locations starting from 0x0206 into myReadBuffer
readID = DRV_PMP_Read ( handle, deviceAddress, &myReadBuffer[0], nBytes);
```

Transfer Status

The status of the read/write transfers can be obtained using API DRV_PMP_TransferStatus.

Example:

```
DRV_PMP_TRANSFER_STATUS writeStatus, readStatus;
uint32_t writeID, readID;

writeStatus = DRV_PMP_TransferStatus( DRV_PMP_INDEX_0, writeID);
readStatus = DRV_PMP_TransferStatus( DRV_PMP_INDEX_0, readID);
```

Client Operation

This section describes general client operation.

Description

General Client Operation

For the application to start using an instance of the module, it must call the DRV_PMP_Open function with a specific intent. This provides the configuration required to open the PMP instance for operation. If the driver is deinitialized using the function DRV_PMP_Deinitialize, the application must call the DRV_PMP_Open function again to set up the instance of the PMP. The function DRV_PMP_Open need not be called again if the system is reinitialized using the DRV_PMP_Reinitialize function.

The PMP driver supports DRV_IO_INTENT_NONBLOCKING, DRV_IO_INTENT_BLOCKING, DRV_IO_INTENT_EXCLUSIVE, and DRV_IO_INTENT_SHARED IO.

Example:

```
DRV_HANDLE handle;

// Open the instance DRV_PMP_INDEX_0 with Non-blocking and Shared intent
handle = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_SHARED | DRV_IO_INTENT_NONBLOCKING);

if( handle == DRV_HANDLE_INVALID )
{
    // Client cannot open the instance.
```

```
}
```

The function DRV_PMP_Close closes an already opened instance of the PMP driver, invalidating the handle. DRV_PMP_Open must have been called to obtain a valid opened device handle.

Example:

```
DRV_HANDLE handle;

// Open the instance DRV_PMP_INDEX_0 with Non-blocking and Shared intent
handle = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_SHARED | DRV_IO_INTENT_NONBLOCKING);

/*...*/

DRV_PMP_Close( handle );
```

The client has the option to check the status through the function DRV_PMP_ClientStatus.

Example:

```
DRV_HANDLE handle;

// Open the instance DRV_PMP_INDEX_0 with Non-blocking and Shared intent
handle = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_SHARED | DRV_IO_INTENT_NONBLOCKING);

if ( DRV_PMP_CLIENT_STATUS_OPEN != DRV_PMP_ClientStatus( handle ) )
   return 0;
```

Client Mode Setting

Any client-specific PMP configuration has to be done using a separate function, DRV_PMP_ModeConfig. This function must be called after the client is open using DRV_PMP_Open.

Following are the client-specific configuration parameters the user can set using this function:

Configuration Parameter	Description
pmpMode	Selects the PMP mode (master or slave) to use.
intMode	Selects the interrupt mode to use.
incrementMode	Sets up address for either auto-increment or decrement mode.
endianMode	Sets Little/Big endian mode.
portSize	Specifies the data width (8-bit or 16-bit).
waitStates	Selects the different wait states.
chipSelect	Selects the Chip Select line.

Example:

```
DRV_HANDLE handle;
DRV_PMP_MODE_CONFIG config;

config.chipSelect = PMCS1_AND_PMCS2_AS_CHIP_SELECT;
config.endianMode = LITTLE_ENDIAN;
config.incrementMode = PMP_ADDRESS_AUTO_INCREMENT;
config.intMode =  PMP_INTERRUPT_NONE;
config.pmpMode = PMP_MASTER_READ_WRITE_STROBES_INDEPENDENT; //Master Mode 2
config.portSize =  PMP_DATA_SIZE_8_BITS;
config.waitStates.dataHoldWait = PMP_DATA_HOLD_2;
config.waitStates.dataWait = PMP_DATA_WAIT_THREE;
config.waitStates.strobeWait = PMP_STROBE_WAIT_5;

// Open the instance DRV_PMP_INDEX_0 with Non-blocking and Shared intent
handle = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_SHARED | DRV_IO_INTENT_NONBLOCKING);

// Configure the client
DRV_PMP_ModeConfig ( handle, config );
```

Example Code for Complete Operation

A code example of complete operation is provided in this section.

Description

This example code will write 100 bytes of data twice (i.e., repeat once) in the memory location starting from 0x0206, and then it will be read in the buffer, myReadBuffer. The modes selected for this transfer are:

- Non-blocking
- No Interrupt
- PMP Master Mode 2
- Address Auto-increment
- No Address/Data Lines Multiplexing
- 8-bit data

Example:

```
void main(void)
{
DRV_PMP_INIT     init;
SYS_MODULE_OBJ   object;
SYS_STATUS       pmpStatus;
DRV_HANDLE       handle;
DRV_PMP_MODE_CONFIG config;
unsigned char myReadBuffer[300], myWriteBuffer[100];
uint32_t deviceAddress, nBytes, repeatCount, i;
uint32_t writeID, readID;
DRV_PMP_TRANSFER_STATUS writeStatus=0, readStatus=0;


// populate the PMP init configuration structure
    init.inputBuffer = PMP_INPUT_BUFFER_TTL;
    init.polarity.addressLatchPolarity = PMP_POLARITY_ACTIVE_HIGH;
    init.polarity.rwStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
    init.polarity.writeEnableStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
    init.polarity.chipselect1Polarity = PMP_POLARITY_ACTIVE_HIGH;
    init.polarity.chipselect2Polarity = PMP_POLARITY_ACTIVE_LOW;
    init.ports.addressPortsMask = PMP_PMA0_PORT | PMP_PMA1_PORT | PMP_PMA2_TO_PMA13_PORTS;
    init.ports.readWriteStrobe = PORT_ENABLE;
    init.ports.writeEnableStrobe = PORT_ENABLE;
    init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
    init.pmpID            = PMP_ID_0;
    init.stopInIdle       = false;
    init.muxMode          = PMP_MUX_NONE;

object = DRV_PMP_Initialize (DRV_PMP_INDEX_0, (SYS_MODULE_INIT *)&init);

pmpStatus = DRV_PMP_Status(object);

if ( SYS_STATUS_READY != pmpStatus)
{
    // Handle error
}

// Open the instance DRV_PMP_INDEX_0 with Non-blocking and Shared intent
handle = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_SHARED | DRV_IO_INTENT_NONBLOCKING);

if( handle == DRV_HANDLE_INVALID )
{
    // Client cannot open the instance.
}

config.chipSelect = PMCS1_AND_PMCS2_AS_CHIP_SELECT;
config.endianMode = LITTLE_ENDIAN;
```

```
config.incrementMode = PMP_ADDRESS_AUTO_INCREMENT;
config.intMode =  PMP_INTERRUPT_NONE;
config.pmpMode = PMP_MASTER_READ_WRITE_STROBES_INDEPENDENT; //Master Mode 2
config.portSize =  PMP_DATA_SIZE_8_BITS;
config.waitStates.dataHoldWait = PMP_DATA_HOLD_2;
config.waitStates.dataWait = PMP_DATA_WAIT_THREE;
config.waitStates.strobeWait = PMP_STROBE_WAIT_5;

// Configure the client
DRV_PMP_ModeConfig ( handle, config );

deviceAddress = 0x0206;
nBytes = 100;
repeatCount = 0x01;
for (i=0; i<nBytes; i++)
{
   myWriteBuffer[i]=i*5+7;
}

writeID = DRV_PMP_Write ( handle, deviceAddress, &myWriteBuffer[0], nBytes, repeatCount);
readID = DRV_PMP_Read ( handle, deviceAddress, &myReadBuffer[0], nBytes*2);


while(!((writeStatus == PMP_TRANSFER_FINISHED)&&(readStatus == PMP_TRANSFER_FINISHED)))
{
    DRV_PMP_Tasks (object);

    writeStatus = DRV_PMP_TransferStatus( DRV_PMP_INDEX_0, writeID);
    readStatus = DRV_PMP_TransferStatus( DRV_PMP_INDEX_0, readID);
}

while(1);
}
```

Configuring the Library

Macros

	Name	Description
	DRV_PMP_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_PMP_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_PMP_QUEUE_SIZE	PMP queue size for different instances.

Description

The configuration of the PMP driver is based on the file `drv_pmp_config.h`.

This header file contains the configuration selection for the PMP Driver. Based on the selections made, the PMP Driver may support the selected features. These configuration settings will apply to all instances of the PMP Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

DRV_PMP_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_pmp_config.h

C

```
#define DRV_PMP_CLIENTS_NUMBER 2
```

Description

PMP maximum number of clients

This definition select the maximum number of clients that the PMP driver can support at run time.

Remarks

None.

DRV_PMP_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver.

File

drv_pmp_config.h

C

```
#define DRV_PMP_INSTANCES_NUMBER 1
```

Description

PMP hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver.

Remarks

None.

DRV_PMP_QUEUE_SIZE Macro

PMP queue size for different instances.

File

drv_pmp_config.h

C

```
#define DRV_PMP_QUEUE_SIZE 8
```

Description

PMP queue size

The PMP queue size for a driver instances should be placed here. If more than one driver instance of PMP is present, then all takes the same queue size.

Remarks

All the transfers (Read/Write) first gets queued and gets completed sequentially when Task API is called in a loop. Therefore, the minimum value of this index should be 1.

Building the Library

This section lists the files that are available in the PMP Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/pmp`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_pmp.h	This file provides the interface definitions of the PMP driver

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_pmp_dynamic.c	This file contains the core implementation of the PMP driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The PMP Driver Library depends on the following modules:

- PMP Peripheral Library
- Interrupt System Service Library

Library Interface

a) System Functions

	Name	Description
	DRV_PMP_Deinitialize	Deinitializes the specified instance of the PMP driver module. **Implementation:** Dynamic
	DRV_PMP_Initialize	Initializes the PMP driver. **Implementation:** Static/Dynamic
	DRV_PMP_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_PMP_Status	Provides the current status of the PMP driver module. **Implementation:** Dynamic
	DRV_PMP_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic
	DRV_PMP_TimingSet	Sets PMP timing parameters. **Implementation:** Static

b) Client Interaction Functions

	Name	Description
	DRV_PMP_ClientStatus	Gets the current client-specific status of the PMP driver. **Implementation:** Dynamic
	DRV_PMP_Close	Closes an opened instance of the PMP driver. **Implementation:** Dynamic
	DRV_PMP_ModeConfig	Configures the PMP modes. **Implementation:** Static/Dynamic
	DRV_PMP_Open	Opens the specified PMP driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_PMP_Read	Read the data from external device. **Implementation:** Static/Dynamic
	DRV_PMP_Write	Transfers the data from the MCU to the external device. **Implementation:** Static/Dynamic

c) Client Transfer Functions

	Name	Description
	DRV_PMP_TransferStatus	Returns the transfer status. **Implementation:** Dynamic

e) Data Types and Constants

	Name	Description
	DRV_PMP_INDEX_COUNT	Number of valid PMP driver indices.
	DRV_PMP_CHIPX_STROBE_MODE	PMP writeEnable/ReadWrite strobes.
	DRV_PMP_CLIENT_STATUS	PMP client status definitions.
	DRV_PMP_ENDIAN_MODE	PMP Endian modes.
	DRV_PMP_INDEX	PMP driver index definitions.
	DRV_PMP_INIT	Defines the PMP driver initialization data.
	DRV_PMP_MODE_CONFIG	PMP modes configuration.
	DRV_PMP_POLARITY_OBJECT	PMP polarity object.
	DRV_PMP_PORT_CONTROL	PMP port enable/disable definitions.

	DRV_PMP_PORTS	PMP port configuration.
	DRV_PMP_QUEUE_ELEMENT_OBJ	Defines the object for PMP queue element.
	_DRV_PMP_QUEUE_ELEMENT_OBJ	Defines the object for PMP queue element.
	DRV_PMP_TRANSFER_STATUS	Defines the PMP transfer status.
	DRV_PMP_WAIT_STATES	PMP wait states object.
	_QUEUE_ELEMENT_OBJECT	Defines the structure required for maintaining the queue element.
	MAX_NONBUFFERED_BYTE_COUNT	After this number the PMP transfer should be polled to guarantee data transfer
	DRV_PMP_TRANSFER_TYPE	This is type DRV_PMP_TRANSFER_TYPE.
	PMP_QUEUE_ELEMENT_OBJECT	Defines the structure required for maintaining the queue element.

Description

This section describes the Application Programming Interface (API) functions of the PMP Driver.

Refer to each section for a detailed description.

a) System Functions

DRV_PMP_Deinitialize Function

Deinitializes the specified instance of the PMP driver module.

Implementation: Dynamic

File

drv_pmp.h

C

```
void DRV_PMP_Deinitialize(const SYS_MODULE_OBJ pmpDriverObject);
```

Returns

None.

Description

This function deinitializes the specified instance of the PMP driver module, disabling its operation (and any hardware). All internal data is invalidated.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_PMP_Status operation. The system has to use DRV_PMP_Status to find out when the module is in the ready state.

Preconditions

The DRV_PMP_Initialize function must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      pmpDriverObject;     //  Returned from DRV_PMP_Initialize
SYS_STATUS          status;

DRV_PMP_Deinitialize(pmpDriverObject);

status = DRV_PMP_Status(pmpDriverObject);
if (SYS_MODULE_DEINITIALIZED == status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
pmpDriverObject	Driver object handle, returned from the DRV_PMP_Initialize

Function

void DRV_PMP_Deinitialize (SYS_MODULE_OBJ pmpDriverObject)

DRV_PMP_Initialize Function

Initializes the PMP driver.

Implementation: Static/Dynamic

File

drv_pmp.h

C

```
SYS_MODULE_OBJ DRV_PMP_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

Returns

If successful, it returns a valid handle to a driver object. Otherwise, it returns SYS_MODULE_OBJ_INVALID. The returned object must be passed as argument to DRV_PMP_Reinitialize, DRV_PMP_Deinitialize, DRV_PMP_Tasks and DRV_PMP_Status routines.

Description

This function initializes the PMP driver, making it ready for clients to open and use it.

Remarks

This function must be called before any other PMP function is called.

This function should only be called once during system initialization unless DRV_PMP_Deinitialize is called to deinitialize the driver instance.

This function will NEVER block for hardware access. If the operation requires time to allow the hardware to reinitialize, it will be reported by the DRV_PMP_Status operation. The system must use DRV_PMP_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
DRV_PMP_INIT    init;
SYS_MODULE_OBJ  objectHandle;

// Populate the initialization structure
init.inputBuffer = PMP_INPUT_BUFFER_TTL;
init.polarity.addressLatchPolarity = PMP_POLARITY_ACTIVE_HIGH;
init.polarity.rwStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
init.polarity.writeEnableStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
init.polarity.chipselect1Polarity = PMP_POLARITY_ACTIVE_HIGH;
init.polarity.chipselect2Polarity = PMP_POLARITY_ACTIVE_LOW;
init.ports.addressPortsMask = PMP_PMA0_PORT | PMP_PMA1_PORT | PMP_PMA2_TO_PMA13_PORTS | PMP_PMA14_PORT;
init.ports.readWriteStrobe = PORT_ENABLE;
init.ports.writeEnableStrobe = PORT_ENABLE;
init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
init.pmpID            = PMP_ID_0;
init.stopInIdle       = false;
init.muxMode          = PMP_MUX_NONE;

// Do something

objectHandle = DRV_PMP_Initialize(DRV_PMP_INDEX_0,  (SYS_MODULE_INIT*)&init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Index for the driver instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the driver

Function

SYS_MODULE_OBJ DRV_PMP_Initialize(const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT * const init)

DRV_PMP_Reinitialize Function

Reinitializes the driver and refreshes any associated hardware settings.

Implementation: Dynamic

File

drv_pmp.h

C

```
void DRV_PMP_Reinitialize(const SYS_MODULE_OBJ pmpDriverObject, const SYS_MODULE_INIT * const
init);
```

Returns

None.

Description

This function reinitializes the driver and refreshes any associated hardware settings using the specified initialization data, but it will not interrupt any ongoing operations.

Remarks

This function can be called multiple times to reinitialize the module.

This operation can be used to refresh any supported hardware registers as specified by the initialization data or to change the power state of the module.

This function will NEVER block for hardware access. If the operation requires time to allow the hardware to re-initialize, it will be reported by the DRV_PMP_Status operation. The system must use DRV_PMP_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

The DRV_PMP_Initialize function must have been called before calling this function and a valid SYS_MODULE_OBJ must have been returned.

Example

```
DRV_PMP_INIT    init;
SYS_MODULE_OBJ  pmpDriverObject;
SYS_STATUS  pmpStatus;

// Populate the initialization structure
init.inputBuffer = PMP_INPUT_BUFFER_TTL;
init.polarity.addressLatchPolarity = PMP_POLARITY_ACTIVE_HIGH;
init.polarity.rwStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
init.polarity.writeEnableStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
init.polarity.chipselect1Polarity = PMP_POLARITY_ACTIVE_HIGH;
init.polarity.chipselect2Polarity = PMP_POLARITY_ACTIVE_LOW;
init.ports.addressPortsMask = PMP_PMA0_PORT | PMP_PMA1_PORT | PMP_PMA2_TO_PMA13_PORTS | PMP_PMA14_PORT;
init.ports.readWriteStrobe = PORT_ENABLE;
init.ports.writeEnableStrobe = PORT_ENABLE;
init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
init.pmpID            = PMP_ID_0;
init.stopInIdle       = false;
init.muxMode          = PMP_MUX_NONE;

DRV_PMP_Reinitialize(pmpDriverObject, (SYS_MODULE_INIT*)&init);

pmpStatus = DRV_PMP_Status(pmpDriverObject);
if (SYS_STATUS_BUSY == pmpStatus)
{
    // Check again later to ensure the driver is ready
}
```

```
else if (SYS_STATUS_ERROR >= pmpStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
pmpDriverObject	Driver object handle, returned from the DRV_PMP_Initialize

Function

void DRV_PMP_Reinitialize (SYS_MODULE_OBJ pmpDriverObject,

const SYS_MODULE_INIT * const init)

init - Pointer to the initialization data structure

DRV_PMP_Status Function

Provides the current status of the PMP driver module.

Implementation: Dynamic

File

drv_pmp.h

C

```
SYS_STATUS DRV_PMP_Status(const SYS_MODULE_OBJ pmpDriverObject);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another

Description

This function provides the current status of the PMP driver module.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_STATUS_BUSY - Indicates that the driver is busy with a previous system level operation and cannot start another

SYS_STATUS_ERROR - Indicates that the driver is in an error state

Any value less than SYS_STATUS_ERROR is also an error state.

SYS_MODULE_DEINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

This operation can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, a previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_PMP_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      pmpDriverObject;     // Returned from DRV_PMP_Initialize
SYS_STATUS          status;

status = DRV_PMP_Status(pmpDriverObject);
else if (SYS_STATUS_ERROR >= status)
{
    // Handle error
}
```

Parameters

Parameters	Description
pmpDriverObject	Driver object handle, returned from the DRV_PMP_Initialize routine

Function

SYS_STATUS DRV_PMP_Status (SYS_MODULE_OBJ pmpDriverObject)

DRV_PMP_Tasks Function

Maintains the driver's state machine and implements its ISR.

Implementation: Dynamic

File

drv_pmp.h

C

```
void DRV_PMP_Tasks(SYS_MODULE_OBJ pmpDriverObject);
```

Returns

None.

Description

This function is used to maintain the queue and execute the tasks stored in the queue. It resides in the ISR of the PMP for interrupt-driven implementations.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This function may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_PMP_Initialize function must have been called for the specified PMP driver instance.

Example

```
SYS_MODULE_OBJ      pmpDriverObject;     // Returned from DRV_PMP_Initialize

while (true)
{
    DRV_PMP_Tasks (pmpDriverObject);

    // Do other tasks
}
```

Parameters

Parameters	Description
pmpDriverObject	Object handle for the specified driver instance (returned from DRV_PMP_Initialize)

Function

void DRV_PMP_Tasks (SYS_MODULE_OBJ pmpDriverObject);

DRV_PMP_TimingSet Function

Sets PMP timing parameters.

Implementation: Static

File

drv_pmp.h

C

```
void DRV_PMP_TimingSet(PMP_DATA_WAIT_STATES dataWait, PMP_STROBE_WAIT_STATES strobeWait,
PMP_DATA_HOLD_STATES dataHold);
```

Returns

None.

Description

This function sets the PMP timing parameters.

Remarks

None.

Preconditions

The DRV_PMP_Initialize function must have been called.

Example

```
DRV_PMP0_TimingSet(PMP_DATA_WAIT_THREE,PMP_STROBE_WAIT_6,PMP_DATA_HOLD_4);
```

Parameters

Parameters	Description
dataWait	Data setup to read/write strobe wait states
strobeWait	Read/write strobe wait states
dataHold	Data hold time after read/write strobe wait states

Function

void DRV_PMP_TimingSet(

PMP_DATA_WAIT_STATES dataWait,

PMP_STROBE_WAIT_STATES strobeWait,

PMP_DATA_HOLD_STATES dataHold

)

b) Client Interaction Functions

DRV_PMP_ClientStatus Function

Gets the current client-specific status of the PMP driver.

Implementation: Dynamic

File

drv_pmp.h

C

```
DRV_PMP_CLIENT_STATUS DRV_PMP_ClientStatus(DRV_HANDLE hClient);
```

Returns

A DRV_PMP_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the PMP driver associated with the specified handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_PMP_Initialize routine must have been called.

DRV_PMP_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE hClient;  // Returned from DRV_PMP_Open
DRV_PMP_CLIENT_STATUS pmpClientStatus;

pmpClientStatus = DRV_PMP_ClientStatus(hClient);
if(DRV_PMP_CLIENT_STATUS_ERROR >= pmpClientStatus)
{
    // Handle the error
}
```

Parameters

Parameters	Description
hClient	A valid open-instance handle, returned from the driver's open routine

Function

DRV_PMP_CLIENT_STATUS DRV_PMP_ClientStatus (DRV_HANDLE hClient)

DRV_PMP_Close Function

Closes an opened instance of the PMP driver.

Implementation: Dynamic

File

drv_pmp.h

C

```
void DRV_PMP_Close(const DRV_HANDLE hClient);
```

Returns

None

Description

This function closes an opened instance of the PMP driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_PMP_Open before the caller may use the driver again.

If DRV_IO_INTENT_BLOCKING was requested and the driver was built appropriately to support blocking behavior call may block until the operation is complete.

If DRV_IO_INTENT_NON_BLOCKING request the driver client can call the DRV_PMP_Status operation to find out when the module is in the ready state (the handle is no longer valid).

Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_PMP_Initialize routine must have been called for the specified PMP driver instance.

DRV_PMP_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE hClient;  // Returned from DRV_PMP_Open

DRV_PMP_Close(hClient);
```

Parameters

Parameters	Description
hClient	A valid open instance handle, returned from the driver's open routine

Function

void DRV_PMP_Close (DRV_HANDLE hClient)

DRV_PMP_ModeConfig Function

Configures the PMP modes.

Implementation: Static/Dynamic

File

drv_pmp.h

C

```
void DRV_PMP_ModeConfig(DRV_HANDLE hClient, DRV_PMP_MODE_CONFIG config);
```

Returns

None.

Description

This function configures the modes for client in which it wants to operate. Different master-slave modes, 8/16 data bits selection, address increment/decrement, interrupt mode, wait states, etc., can be configured through this function.

Remarks

This function will NEVER block waiting for hardware. If this API is called more than once for a particular client handle, previous config setting of that client will be overwritten.

Preconditions

Function DRV_PMP_Initialize must have been called. DRV_PMP_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE  hClient;
DRV_PMP_MODE_CONFIG config;

config.chipSelect = PMCS1_AND_PMCS2_AS_CHIP_SELECT;
config.endianMode = LITTLE_ENDIAN;
config.incrementMode = PMP_ADDRESS_AUTO_INCREMENT;
config.intMode =  PMP_INTERRUPT_NONE;
config.pmpMode = PMP_MASTER_READ_WRITE_STROBES_INDEPENDENT;
config.portSize =  PMP_DATA_SIZE_8_BITS;
config.waitStates.dataHoldWait = PMP_DATA_HOLD_2;
config.waitStates.dataWait = PMP_DATA_WAIT_THREE;
config.waitStates.strobeWait = PMP_STROBE_WAIT_5;

DRV_PMP_ModeConfig ( hClient, config );
```

Parameters

Parameters	Description
hClient	Client handle obtained from DRV_PMP_Open API
config	Structure which will have all the required PMP modes configuration

Function

void DRV_PMP_ModeConfig (DRV_HANDLE hClient,
                              DRV_PMP_MODE_CONFIG config)

DRV_PMP_Open Function

Opens the specified PMP driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_pmp.h

C

```
DRV_HANDLE DRV_PMP_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID.

Description

This function opens the specified PMP driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_PMP_Close routine is called.

This function will NEVER block waiting for hardware.

If the DRV_IO_INTENT_BLOCKING is requested and the driver was built appropriately to support blocking behavior, other client-level operations may block waiting on hardware until they are complete.

If DRV_IO_INTENT_NON_BLOCKING is requested the driver client can call the DRV_PMP_ClientStatus operation to find out when the module is in the ready state.

If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID.

Preconditions

The DRV_PMP_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  hClient;

hClient = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == hClient)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_PMP_Open (const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT intent)

DRV_PMP_Read Function

Read the data from external device.

Implementation: Static/Dynamic

File

drv_pmp.h

C

```
PMP_QUEUE_ELEMENT_OBJECT* DRV_PMP_Read(DRV_HANDLE hClient, uint32_t address, uint16_t* buffer,
uint32_t nBytes);
```

Returns

Returns the position number of the queue, where the data element was stored. Returns '0' when there is no place in the queue to store the data.

Description

This function reads the given number of data bytes from the given address of the external device to the MCU buffer through the selected PMP instance. This function should be used for all the master and slave modes. Proper configuration should be done using DRV_PMP_ModeConfig before calling this function.

Preconditions

The DRV_PMP_Initialize routine must have been called. DRV_PMP_Open must have been called to obtain a valid opened device handle. DRV_PMP_ModeConfig must have been called to configure the desired mode

Example

```
DRV_HANDLE hClient;  // Returned from DRV_PMP_Open
uint32_t deviceAddress;
uint32_t nBytes;
unsigned char myBuffer[nBytes];
uint32_t transferID;

transferID = DRV_PMP_Read ( hClient, deviceAddress, &myBuffer, nBytes);
```

Parameters

Parameters	Description
hClient	A valid open-instance handle, returned from the driver's open routine
address	Starting address of the slave device from where data has to be read. It does not have any significance for legacy slave mode and buffer mode. In PMP enhanced slave mode i.e. addressable buffer slave mode, this parameter should be the buffer number to be used.
buffer	Pointer to the buffer into which the data read through the PMP instance will be placed. Even if only one word has to be transferred, pointer should be used.
nBytes	Number of bytes that need to be read through the PMP instance

Function

uint32_t DRV_PMP_Read (DRV_HANDLE hClient,

uint32_t address,

unsigned char* buffer,

uint32_t nBytes)

DRV_PMP_Write Function

Transfers the data from the MCU to the external device.

Implementation: Static/Dynamic

File

drv_pmp.h

C

```
PMP_QUEUE_ELEMENT_OBJECT* DRV_PMP_Write(DRV_HANDLE* hClient, bool address, uint32_t * buffer,
uint32_t nBytes, uint32_t repeatCount);
```

Returns

Returns a 32-bit ID with which status of the transfer can be checked later. Returns '0' when there is no place in the queue to store the data.

Description

This function transfer the given number of data bytes from the MCU buffer location to the defined address of the external device through the selected PMP instance. It repeats the operation n (=repeatCount) number of times as well. This function should be used for all the master and slave modes. Proper configuration should be done using DRV_PMP_ModeConfig before calling this function.

Preconditions

The DRV_PMP_Initialize routine must have been called. DRV_PMP_Open must have been called to obtain a valid opened device handle. DRV_PMP_ModeConfig must have been called to configure the desired mode.

Example

```
DRV_HANDLE hClient;  // Returned from DRV_PMP_Open
uint32_t deviceAddress;
uint32_t nBytes;
unsigned char myBuffer[nBytes];
uint32_t repeatCount;
uint32_t transferID;

transferID = DRV_PMP_MasterWrite ( hClient, deviceAddress, &myBuffer, nBytes, repeatCount);
```

Parameters

Parameters	Description
hClient	A valid open-instance handle, returned from the driver's open routine
address	Starting address of the slave device where data has to be written. It does not have any significance for legacy slave mode and buffer mode. In PMP enhanced slave mode (i.e., addressable buffer slave mode), this parameter should be the buffer number to be used.
buffer	Pointer to MCU Buffer from which the data will be written through the PMP instance. even if only one word has to be transferred, pointer should be used.
nBytes	Total number of bytes that need to be written through the PMP instance
repeatCount	Number of times the data set (nBytes of data) to be repeatedly written. This value should be 0 if user does not want any repetition. If repeatCount is greater than 0, then after writing every nBytes of data, the buffer starts pointing to its first element. Ideally, PMP Address should be in auto increment/decrement mode for repeatCount greater than 0.

Function

uint32_t DRV_PMP_Write (DRV_HANDLE hClient,

uint32_t address,

```
unsigned char* buffer,
uint32_t nBytes,
uint32_t repeatCount)
```

c) Client Transfer Functions

DRV_PMP_TransferStatus Function

Returns the transfer status.

Implementation: Dynamic

File

drv_pmp.h

C

```
DRV_PMP_TRANSFER_STATUS DRV_PMP_TransferStatus(PMP_QUEUE_ELEMENT_OBJECT* queueObject);
```

Returns

A DRV_PMP_TRANSFER_STATUS value describing the current status of the transfer.

Description

This function returns the status of a particular transfer whose ID has been specified as input.

Example

```
uint32_8 seqID;
DRV_PMP_TRANSFER_STATUS transferStatus;

transferStatus = DRV_PMP_TransferStatus( DRV_PMP_INDEX_0, seqID);
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
seqID	A valid ID returned from read/write transfer functions

Function

DRV_PMP_TRANSFER_STATUS DRV_PMP_TransferStatus(DRV_HANDLE hClient)

d) Miscellaneous Functions

e) Data Types and Constants

DRV_PMP_INDEX_COUNT Macro

Number of valid PMP driver indices.

File

drv_pmp.h

C

```
#define DRV_PMP_INDEX_COUNT _PMP_EXISTS
```

Description

PMP Driver Module Index Count

This constant identifies the number of valid PMP driver indices.

Remarks

The value of "_PMP_EXISTS" is derived from device-specific header files defined as part of the peripheral libraries.

DRV_PMP_CHIPX_STROBE_MODE Enumeration

PMP writeEnable/ReadWrite strobes.

File

drv_pmp.h

C

```
typedef enum {
  PMP_RW_STROBE_WITH_ENABLE_STROBE,
  PMP_READ_AND_WRITE_STROBES
} DRV_PMP_CHIPX_STROBE_MODE;
```

Members

Members	Description
PMP_RW_STROBE_WITH_ENABLE_STROBE	One strobe for read/write and another for enable
PMP_READ_AND_WRITE_STROBES	Separate strobes for read and write operations

Description

PMP writeEnable/ReadWrite strobes

This enumeration provides ReadWrite/WriteEnable Strobe definitions.

DRV_PMP_CLIENT_STATUS Enumeration

PMP client status definitions.

File

drv_pmp.h

C

```
typedef enum {
  DRV_PMP_CLIENT_STATUS_INVALID,
  PMP_CLIENT_STATUS_CLOSED,
  DRV_PMP_CLIENT_STATUS_OPEN
} DRV_PMP_CLIENT_STATUS;
```

Description

PMP Client Status

This enumeration provides various client status possibilities.

DRV_PMP_ENDIAN_MODE Enumeration

PMP Endian modes.

File

drv_pmp.h

C

```
typedef enum {
  LITTLE,
  BIG
} DRV_PMP_ENDIAN_MODE;
```

Members

Members	Description
LITTLE	Little Endian
BIG	Big Endian

Description

PMP Endian modes

This enumeration holds the Endian configuration options.

DRV_PMP_INDEX Enumeration

PMP driver index definitions.

File

drv_pmp.h

C

```
typedef enum {
  DRV_PMP_INDEX_0,
  DRV_PMP_INDEX_1
} DRV_PMP_INDEX;
```

Members

Members	Description
DRV_PMP_INDEX_0	First PMP instance
DRV_PMP_INDEX_1	Second PMP instance (not available for now)

Description

PMP Driver Module Index Numbers

These constants provide PMP driver index definitions.

Remarks

These values should be passed into the DRV_PMP_Initialize and DRV_PMP_Open functions to identify the driver instance in use.

DRV_PMP_INIT Structure

Defines the PMP driver initialization data.

File

drv_pmp.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  PMP_MODULE_ID pmpID;
  bool stopInIdle;
  PMP_MUX_MODE muxMode;
  PMP_INPUT_BUFFER_TYPE inputBuffer;
  DRV_PMP_POLARITY_OBJECT polarity;
  DRV_PMP_PORTS ports;
} DRV_PMP_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	module power state info
PMP_MODULE_ID pmpID;	module PLIB ID
bool stopInIdle;	Stop in Idle enable
PMP_MUX_MODE muxMode;	MUX mode
PMP_INPUT_BUFFER_TYPE inputBuffer;	Input buffer type to be used
DRV_PMP_POLARITY_OBJECT polarity;	Polarity settings
DRV_PMP_PORTS ports;	PMP port settings

Description

PMP Driver Initialize Data

This data type defines data required to initialize or reinitialize the PMP driver.

Remarks

Not all the initialization features are available for all devices.

DRV_PMP_MODE_CONFIG Structure

PMP modes configuration.

File

drv_pmp.h

C

```
typedef struct {
  PMP_OPERATION_MODE pmpMode;
  PMP_INTERRUPT_MODE intMode;
  PMP_INCREMENT_MODE incrementMode;
  DRV_PMP_ENDIAN_MODE endianMode;
  PMP_DATA_SIZE portSize;
  DRV_PMP_WAIT_STATES waitStates;
  PMP_CHIPSELECT_FUNCTION chipSelect;
} DRV_PMP_MODE_CONFIG;
```

Members

Members	Description
PMP_OPERATION_MODE pmpMode;	PMP Usage Mode Type
PMP_INTERRUPT_MODE intMode;	Interrupt mode
PMP_INCREMENT_MODE incrementMode;	should be appropriately selected based on read/write requirements and operation mode setting */ address/buffer increment mode
DRV_PMP_ENDIAN_MODE endianMode;	it does not have any significance in PMP slave mode or 8bit data mode */ Endian modes
PMP_DATA_SIZE portSize;	Data Port Size
DRV_PMP_WAIT_STATES waitStates;	Wait states
PMP_CHIPSELECT_FUNCTION chipSelect;	use this when PLIB is fixed

Description

PMP modes configuration

This data type controls the configuration of PMP modes.

DRV_PMP_POLARITY_OBJECT Structure

PMP polarity object.

File

drv_pmp.h

C

```
typedef struct {
  PMP_POLARITY_LEVEL addressLatchPolarity;
  PMP_POLARITY_LEVEL byteEnablePolarity;
  PMP_POLARITY_LEVEL rwStrobePolarity;
  PMP_POLARITY_LEVEL writeEnableStrobePolarity;
  PMP_POLARITY_LEVEL chipselect1Polarity;
  PMP_POLARITY_LEVEL chipselect2Polarity;
} DRV_PMP_POLARITY_OBJECT;
```

Members

Members	Description
PMP_POLARITY_LEVEL addressLatchPolarity;	Address latch polarity
PMP_POLARITY_LEVEL byteEnablePolarity;	ByteEnable port polarity
PMP_POLARITY_LEVEL rwStrobePolarity;	Read/Write strobe polarity
PMP_POLARITY_LEVEL writeEnableStrobePolarity;	Write/Enable strobe polarity
PMP_POLARITY_LEVEL chipselect1Polarity;	ChipSelect-1 Polarity
PMP_POLARITY_LEVEL chipselect2Polarity;	chipSelect-2 Polarity

Description

PMP polarity object

This structure holds the polarities of different entities to be configured.

DRV_PMP_PORT_CONTROL Enumeration

PMP port enable/disable definitions.

File

drv_pmp.h

C

```
typedef enum {
  PORT_ENABLE,
  PORT_DISABLE
} DRV_PMP_PORT_CONTROL;
```

Members

Members	Description
PORT_ENABLE	Enable the given port
PORT_DISABLE	Disable the given port

Description

PMP port enable/disable.

This enumeration provides port enable/disable values.

DRV_PMP_PORTS Structure

PMP port configuration.

File

drv_pmp.h

C

```
typedef struct {
  PMP_ADDRESS_PORT addressPortsMask;
  PMP_PMBE_PORT byteEnablePort;
  DRV_PMP_PORT_CONTROL readWriteStrobe;
  DRV_PMP_PORT_CONTROL writeEnableStrobe;
} DRV_PMP_PORTS;
```

Members

Members	Description
PMP_ADDRESS_PORT addressPortsMask;	User needs to put the address lines which he wants to use in ORed fashion * Address ports
PMP_PMBE_PORT byteEnablePort;	Byte enable ports
DRV_PMP_PORT_CONTROL readWriteStrobe;	READ/WRITE Strobe PORT
DRV_PMP_PORT_CONTROL writeEnableStrobe;	WRITE/ENABLE strobe port

Description

PMP Ports

This structure holds the ports (including the address ports) to be configured by the application to function as general purpose I/O (GPIO) or part of the PMP.

DRV_PMP_QUEUE_ELEMENT_OBJ Structure

Defines the object for PMP queue element.

File

drv_pmp.h

C

```
typedef struct _DRV_PMP_QUEUE_ELEMENT_OBJ {
  struct _DRV_PMP_CLIENT_OBJ * hClient;
  uint32_t buffer;
  uint16_t* addressBuffer;
  uint32_t nTransfers;
  int32_t nRepeats;
  DRV_PMP_TRANSFER_TYPE type;
} DRV_PMP_QUEUE_ELEMENT_OBJ;
```

Members

Members	Description
struct _DRV_PMP_CLIENT_OBJ * hClient;	handle of the client object returned from open API
uint32_t buffer;	pointer to the buffer holding the transmitted data
uint16_t* addressBuffer;	pointer to the buffer holding the transmitted data
uint32_t nTransfers;	number of bytes to be transferred
int32_t nRepeats;	number of times the data set has to be transferred repeatedly
DRV_PMP_TRANSFER_TYPE type;	PMP Read or Write

Description

PMP Driver Queue Element Object

This defines the object structure for each queue element of PMP. This object gets created for every Read/Write operations APIs.

Remarks

None

DRV_PMP_TRANSFER_STATUS Enumeration

Defines the PMP transfer status.

File

drv_pmp.h

C

```
typedef enum {
  MASTER_8BIT_TRANSFER_IN_PROGRESS = PMP_DATA_SIZE_8_BITS,
  MASTER_16BIT_TRANSFER_IN_PROGRESS = PMP_DATA_SIZE_16_BITS,
  MASTER_8BIT_BUFFER_IN_PROGRESS,
  MASTER_16BIT_BUFFER_IN_PROGRESS,
  MASTER_8BIT_TRANSFER_CONTINUE,
  MASTER_8BIT_BUFFER_CONTINUE,
  QUEUED_BUT_PMP_TRANSFER_NOT_STARTED,
  PMP_TRANSFER_FINISHED
} DRV_PMP_TRANSFER_STATUS;
```

Description

Queue Element Transfer Status

This enumeration defines the PMP transfer status.

DRV_PMP_WAIT_STATES Structure

PMP wait states object.

File

drv_pmp.h

C

```
typedef struct {
  PMP_DATA_HOLD_STATES dataHoldWait;
  PMP_STROBE_WAIT_STATES strobeWait;
  PMP_DATA_WAIT_STATES dataWait;
} DRV_PMP_WAIT_STATES;
```

Members

Members	Description
PMP_DATA_HOLD_STATES dataHoldWait;	data hold wait states
PMP_STROBE_WAIT_STATES strobeWait;	read/write strobe wait states
PMP_DATA_WAIT_STATES dataWait;	data wait strobe wait sates

Description

PMP wait states object

This structure holds the different wait states to be configured. Refer to the PMP PLIB help document for the possible values and meaning of the different wait states.

MAX_NONBUFFERED_BYTE_COUNT Macro

File

drv_pmp.h

C

```
#define MAX_NONBUFFERED_BYTE_COUNT 4
/*************************************************************************
                                        After this number the PMP transfer should be polled to
guarantee data
                                        transfer

                                        ***********************************************************
*************/
```

Description

After this number the PMP transfer should be polled to guarantee data transfer

DRV_PMP_TRANSFER_TYPE Enumeration

File

drv_pmp.h

C

```
typedef enum {
  ADDRESS,
  READ,
  WRITE,
  BUFFERED_WRITE
} DRV_PMP_TRANSFER_TYPE;
```

Members

Members	Description
ADDRESS	PMP Address needs to be updated
READ	PMP Read Transfer
WRITE	PMP Write Transfer
BUFFERED_WRITE	PMP Array Write Transfer

Description

This is type DRV_PMP_TRANSFER_TYPE.

PMP_QUEUE_ELEMENT_OBJECT Structure

Defines the structure required for maintaining the queue element.

File

drv_pmp.h

C

```
typedef struct _QUEUE_ELEMENT_OBJECT {
  DRV_PMP_QUEUE_ELEMENT_OBJ data;
  DRV_PMP_TRANSFER_STATUS eTransferStatus;
  uint32_t nTransfersDone;
} PMP_QUEUE_ELEMENT_OBJECT;
```

Members

Members	Description
DRV_PMP_QUEUE_ELEMENT_OBJ data;	The PMP Q Element
DRV_PMP_TRANSFER_STATUS eTransferStatus;	Flag to indicate that the element is in use
uint32_t nTransfersDone;	sequence id

Description

Queue Element Object

This defines the structure required for maintaining the queue element.

Remarks

None

Files

Files

Name	Description
drv_pmp.h	Parallel Master Port (PMP) device driver interface file.
drv_pmp_config.h	PMP driver configuration definitions template

Description

This section lists the source and header files used by the PMP Driver Library.

drv_pmp.h

Parallel Master Port (PMP) device driver interface file.

Enumerations

	Name	Description
	DRV_PMP_CHIPX_STROBE_MODE	PMP writeEnable/ReadWrite strobes.
	DRV_PMP_CLIENT_STATUS	PMP client status definitions.
	DRV_PMP_ENDIAN_MODE	PMP Endian modes.
	DRV_PMP_INDEX	PMP driver index definitions.
	DRV_PMP_PORT_CONTROL	PMP port enable/disable definitions.
	DRV_PMP_TRANSFER_STATUS	Defines the PMP transfer status.
	DRV_PMP_TRANSFER_TYPE	This is type DRV_PMP_TRANSFER_TYPE.

Functions

	Name	Description
	DRV_PMP_ClientStatus	Gets the current client-specific status of the PMP driver. **Implementation:** Dynamic
	DRV_PMP_Close	Closes an opened instance of the PMP driver. **Implementation:** Dynamic
	DRV_PMP_Deinitialize	Deinitializes the specified instance of the PMP driver module. **Implementation:** Dynamic
	DRV_PMP_Initialize	Initializes the PMP driver. **Implementation:** Static/Dynamic
	DRV_PMP_ModeConfig	Configures the PMP modes. **Implementation:** Static/Dynamic
	DRV_PMP_Open	Opens the specified PMP driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_PMP_Read	Read the data from external device. **Implementation:** Static/Dynamic
	DRV_PMP_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_PMP_Status	Provides the current status of the PMP driver module. **Implementation:** Dynamic
	DRV_PMP_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic
	DRV_PMP_TimingSet	Sets PMP timing parameters. **Implementation:** Static

	DRV_PMP_TransferStatus	Returns the transfer status. **Implementation:** Dynamic
	DRV_PMP_Write	Transfers the data from the MCU to the external device. **Implementation:** Static/Dynamic

Macros

	Name	Description
	DRV_PMP_INDEX_COUNT	Number of valid PMP driver indices.
	MAX_NONBUFFERED_BYTE_COUNT	After this number the PMP transfer should be polled to guarantee data transfer

Structures

	Name	Description
	_DRV_PMP_QUEUE_ELEMENT_OBJ	Defines the object for PMP queue element.
	_QUEUE_ELEMENT_OBJECT	Defines the structure required for maintaining the queue element.
	DRV_PMP_INIT	Defines the PMP driver initialization data.
	DRV_PMP_MODE_CONFIG	PMP modes configuration.
	DRV_PMP_POLARITY_OBJECT	PMP polarity object.
	DRV_PMP_PORTS	PMP port configuration.
	DRV_PMP_QUEUE_ELEMENT_OBJ	Defines the object for PMP queue element.
	DRV_PMP_WAIT_STATES	PMP wait states object.
	PMP_QUEUE_ELEMENT_OBJECT	Defines the structure required for maintaining the queue element.

Description

PMP Device Driver Interface

The PMP device driver provides a simple interface to manage the Parallel Master and Slave ports. This file defines the interface definitions and prototypes for the PMP driver.

File Name

drv_pmp.h

Company

Microchip Technology Inc.

drv_pmp_config.h

PMP driver configuration definitions template

Macros

	Name	Description
	DRV_PMP_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_PMP_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_PMP_QUEUE_SIZE	PMP queue size for different instances.

Description

PMP Driver Configuration Definitions for the Template Version

These definitions statically define the driver's mode of operation.

File Name

drv_pmp_config_template.h

Company

Microchip Technology Inc.

RTCC Driver Library

This topic describes the RTCC Driver Library.

Introduction

The Real-Time Clock Calendar (RTCC) Static Driver provides a high-level interface to manage the RTCC module on the Microchip family of microcontrollers.

Description

Through the MHC, this driver provides APIs for the following:

- Initializing the module
- Starting/Stopping the RTCC
- Status functions to yield the date/time
- Status functions to yield the alarm date/time
- Clock output control

Library Interface

Functions

	Name	Description
	DRV_RTCC_AlarmDateGet	Gets the Alarm Date of the RTCC. **Implementation:** Static
	DRV_RTCC_AlarmTimeGet	Gets the Alarm Time of the RTCC. **Implementation:** Static
	DRV_RTCC_ClockOutput	Enables Clock Output for the RTCC. **Implementation:** Static
	DRV_RTCC_DateGet	Gets the Date of the RTCC. **Implementation:** Static
	DRV_RTCC_Initialize	Initializes the RTCC instance for the specified driver index. **Implementation:** Static
	DRV_RTCC_Start	Starts the RTCC. **Implementation:** Static
	DRV_RTCC_Stop	Stops the RTCC. **Implementation:** Static
	DRV_RTCC_TimeGet	Gets the time of the RTCC. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the RTCC Driver Library.

Functions

DRV_RTCC_AlarmDateGet Function

Gets the Alarm Date of the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
uint32_t DRV_RTCC_AlarmDateGet();
```

Returns

uint32_t alarm date value

Description

This routine gets the RTCC alarm date.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

uint32_t DRV_RTCC_AlarmDateGet(void)

DRV_RTCC_AlarmTimeGet Function

Gets the Alarm Time of the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
uint32_t DRV_RTCC_AlarmTimeGet();
```

Returns

uint32_t alarm time value

Description

This routine gets the RTCC alarm time.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

uint32_t DRV_RTCC_AlarmTimeGet(void)

DRV_RTCC_ClockOutput Function

Enables Clock Output for the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
void DRV_RTCC_ClockOutput();
```

Returns

None.

Description

This routine enables the clock output for the RTCC

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

void DRV_RTCC_ClockOutput(void)

DRV_RTCC_DateGet Function

Gets the Date of the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
uint32_t DRV_RTCC_DateGet();
```

Returns

uint32_t date value

Description

This routine gets the RTCC date.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

uint32_t DRV_RTCC_DateGet(void)

DRV_RTCC_Initialize Function

Initializes the RTCC instance for the specified driver index.

Implementation: Static

File

help_drv_rtcc.h

C

```
void DRV_RTCC_Initialize();
```

Returns

None.

Description

This routine initializes the RTCC driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

This routine must be called before any other RTCC routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_RTCC_Initialize(void)

DRV_RTCC_Start Function

Starts the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
void DRV_RTCC_Start();
```

Returns

None.

Description

This routine starts the RTCC, making it ready for clients to use it.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

void DRV_RTCC_Start(void)

DRV_RTCC_Stop Function

Stops the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
void DRV_RTCC_Stop();
```

Returns

None.

Description

This routine stops the RTCC.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

void DRV_RTCC_Stop(void)

DRV_RTCC_TimeGet Function

Gets the time of the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
uint32_t DRV_RTCC_TimeGet();
```

Returns

uint32_t time value

Description

This routine gets the RTCC time.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

uint32_t DRV_RTCC_TimeGet(void)

Secure Digital (SD) Card Driver Library

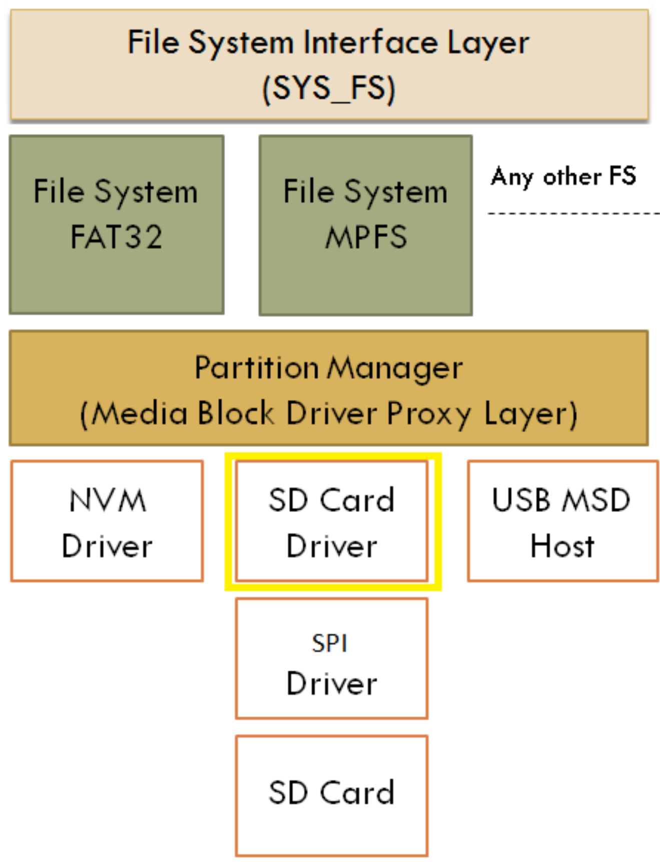
This topic describes the Secure Digital (SD) Card Driver Library.

Introduction

The SD Card driver provides the necessary interfaces to interact with an SD card. It provides the necessary abstraction for the higher layer.

Description

A SD Card is a non-volatile memory (Flash memory) card designed to provide high-capacity memory in a small size. Its applications include digital video camcorders, digital cameras, handheld computers, audio players, and mobile phones.



Using the Library

This topic describes the basic architecture of the SD Card Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_sdcard.h`

The interface to the SD Card Driver library is defined in the `drv_sdcard.h` header file. This file is included by the `drv.h` file. Any C language source (`.c`) file that uses the SD Card Driver library should include `drv.h`.

Please refer to the Understanding MPLAB Harmony section for how the Driver interacts with the framework.

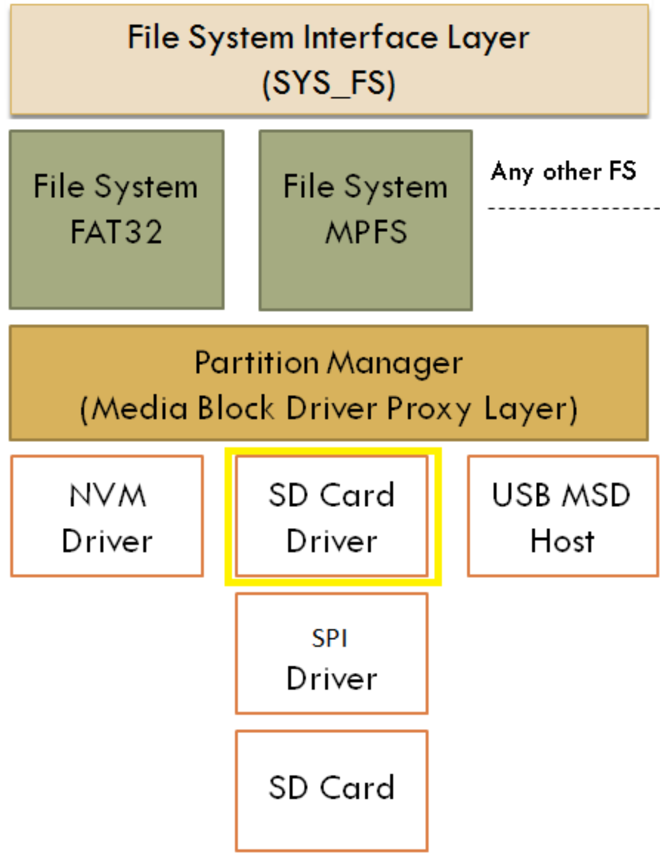
Abstraction Model

This library provides a low-level abstraction of the SD Card Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The SD Card driver comes in the layer below the Partition Manager in the MPLAB Harmony file system architecture and it uses the SPI Driver to interact with the SD card.

SD Card Driver Software Abstraction Block Diagram



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the SD Card module.

Library Interface Section	Description
System Level Functions	Includes functions for initialize the module.
Client Level Functions	Functions to open and close a client.
Operation Functions	Functions for read and write operations
Module Information Functions	Functions for information about the module.
Version Information Functions	Functions to get the software version.

How the Library Works

This section describes how the SD Card Driver Library operates.

Description

Note: Not all modes are available on all devices. Please refer to the specific device data sheet to determine the supported modes.

The library provides interfaces that support:

- System Functionality
- Client Functionality
- Status Functionality

SD Card Driver Initialization

This section provides information for system initialization and reinitialization.

Description

The system performs the initialization and the reinitialization of the device driver with settings that affect only the instance of the device that is being initialized or reinitialized. During system initialization each instance of the SD Card module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_SDCARD_INIT or by using initialization overrides) that are supported by the specific SD Card device hardware:

- SPI Peripheral ID: Identifies the SPI Peripheral ID to be used for the SD Card Driver
- SPI Index: SPI Driver Index
- SD Card frequency: SD Card communication speed
- SPI Clock source: Peripheral clock used by the SPI
- Write-Protect Port: Port used to check if the SD Card is write protected
- Write-Protect Pin: Pin used to check if the SD Card is write protected
- Chip Select Port: Port used for the SPI Chip Select
- Chip Select Pin: Pin used for the SPI Chip Select

The DRV_SDCARD_Initialize function returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the initialize interface would be used by the other system interfaces, such as DRV_SDCARD_Deinitialize, DRV_SDCARD_Status, and DRV_SDCARD_Tasks.

Note: The system initialization and the reinitialization settings, only affect the instance of the peripheral that is being initialized or reinitialized.

Example:

```
const DRV_SDCARD_INIT drvSDCardInit =
{
.spiId = SPI_ID_2,
.spiIndex = 0,
.sdcardSpeedHz = 20000000,
.spiClk = CLK_BUS_PERIPHERAL_2,
.writeProtectPort = PORT_CHANNEL_F,
.writeProtectBitPosition = PORTS_BIT_POS_1,
.chipSelectPort = PORT_CHANNEL_B,
.chipSelectBitPosition = PORTS_BIT_POS_14,
};

void SYS_Initialize (void *data)
{
.
.
sysObj.drvSDCard = DRV_SDCARD_Initialize(DRV_SDCARD_INDEX_0,(SYS_MODULE_INIT *)&drvSDCardInit);
.
.
}
```

Tasks Routine

The system will call DRV_SDCARD_Tasks, from system task service to maintain the driver's state machine.

Client Access Operation

This section provides information for general client operation.

Description

General Client Operation

For the application to start using an instance of the module, it must call the DRV_SDCARD_Open function. This provides the configuration required to open the SD Card instance for operation. If the driver is deinitialized using the function DRV_SDCARD_Deinitialize, the application must call the DRV_SDCARD_Open function again to set up the instance of the SDCARD.

For the various options available for I/O INTENT please refer to Data Types and Constants in the Library Interface section.

Example:

```
DRV_HANDLE handle;
handle = DRV_SDCARD_Open(DRV_SDCARD_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
// Unable to open the driver
}
```

Client Block Data Operation

This topic provides information on client block data operation.

Description

The SDCARD Driver provides a block interface to access the SD Card. The interface provides functionality to read from and write to the SD Card.

Reading Data from the SD Card:

The following steps outline the sequence to be followed for reading data from the SD Card:

1. The system should have completed necessary initialization and DRV_SDCARD_Tasks should either be running in a polled environment, or in an interrupt environment.
2. The driver should have been opened with the necessary intent.
3. Invoke the DRV_SDCARD_Read function and pass the pointer where the data is to be stored, block start address and the number of blocks of data to be read.
4. The client should validate the command handle returned by the DRV_SDCARD_Read function. DRV_SDCARD_COMMAND_HANDLE_INVALID value indicates that an error has occurred which the client needs to handle.
5. If the request was successful then the client can check the status of the request by invoking the DRV_SDCARD_CommandStatus and passing the command handle returned by the read request. Alternately the client could use the event handler for notifications from the driver.
6. The client will be able to close itself by calling the DRV_SDCARD_Close.

Example:

```
// This code shows how to read data from the SD Card
DRV_HANDLE sdcardHandle;
DRV_SDCARD_COMMAND_HANDLE sdcardCommandHandle;
DRV_SDCARD_COMMAND_STATUS commandStatus;
uint8_t readBuf[512];
uint32_t blockAddress;
uint32_t nBlocks;

/* Initialize the block start address and the number of blocks to be read */
blockAddress = 0;
nBlocks = 1;

DRV_SDCARD_Read(sdcardHandle, &sdcardCommandHandle, (uint8_t *)readBuf, blockAddress, nBlocks);
if(DRV_SDCARD_COMMAND_HANDLE_INVALID == sdcardCommandHandle)
{
    /* Failed to queue the read request. Handle the error. */
}
// Wait until the command completes. This should not
// be a while loop if part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
commandStatus = DRV_SDCARD_CommandStatus(sdcardHandle, sdcardCommandHandle);
if(DRV_SDCARD_COMMAND_COMPLETED == commandStatus)
{
    /* Read completed */
}
else if (DRV_SDCARD_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Read Failed */
}
```

Writing Data to the SD Card:

The following steps outline the sequence to be followed for writing data to the SD Card:

1. The system should have completed necessary initialization and DRV_SDCARD_Tasks should either be running in a polled environment, or in an interrupt environment.
2. The driver should have been opened with the necessary intent.
3. Invoke the DRV_SDCARD_Write function and pass the pointer to the data to be written, block start address and the number of blocks of data to be written.

4. The client should validate the command handle returned by the DRV_SDCARD_Write function. DRV_SDCARD_COMMAND_HANDLE_INVALID value indicates that an error has occurred which the client needs to handle.
5. If the request was successful then the client can check the status of the request by invoking the DRV_SDCARD_CommandStatus and passing the command handle returned by the write request. Alternately, the client could use the event handler for notifications from the driver.
6. The client will be able to close itself by calling the DRV_SDCARD_Close.

Example:

```
// This code shows how to write data to the SD Card
DRV_HANDLE sdcardHandle;
DRV_SDCARD_COMMAND_HANDLE sdcardCommandHandle;
DRV_SDCARD_COMMAND_STATUS commandStatus;
uint8_t writeBuf[512];
uint32_t blockAddress;
uint32_t nBlocks;

/* Initialize the block start address and the number of blocks to be written */
blockAddress = 0;
nBlocks = 1;
/* Populate writeBuf with the data to be written */

DRV_SDCARD_Write(sdcardHandle, &sdcardCommandHandle, (uint8_t *)writeBuf, blockAddress, nBlocks);
if(DRV_SDCARD_COMMAND_HANDLE_INVALID == sdcardCommandHandle)
{
    /* Failed to queue the write request. Handle the error. */
}
// Wait until the command completes. This should not
// be a while loop if part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
commandStatus = DRV_SDCARD_CommandStatus(sdcardHandle, sdcardCommandHandle);
if(DRV_SDCARD_COMMAND_COMPLETED == commandStatus)
{
    /* Write completed */
}
else if (DRV_SDCARD_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Write Failed */
}
```

Configuring the Library

Macros

	Name	Description
	DRV_SDCARD_CLIENTS_NUMBER	Selects the miximum number of clients
	DRV_SDCARD_INDEX_MAX	SD Card Static Index selection
	DRV_SDCARD_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver
	DRV_SDCARD_POWER_STATE	Defines an override of the power state of the SD Card driver.

Description

The configuration of the SD Card Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the SD Card Driver. Based on the selections made, the SD Card Driver may support the selected features. These configuration settings will apply to all instances of the SD Card.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

DRV_SDCARD_CLIENTS_NUMBER Macro

Selects the miximum number of clients

File

drv_sdcard_config_template.h

C

```
#define DRV_SDCARD_CLIENTS_NUMBER 1
```

Description

SD Card Maximum Number of Clients

This definition select the maximum number of clients that the SD Card driver can support at run time. Not defining it means using a single client.

Remarks

None.

DRV_SDCARD_INDEX_MAX Macro

SD Card Static Index selection

File

drv_sdcard_config_template.h

C

```
#define DRV_SDCARD_INDEX_MAX 1
```

Description

SD Card Static Index Selection

SD Card Static Index selection for the driver object reference

Remarks

This index is required to make a reference to the driver object

DRV_SDCARD_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver

File

drv_sdcard_config_template.h

C

```
#define DRV_SDCARD_INSTANCES_NUMBER 1
```

Description

SD Card hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver. Not defining it means using a static driver.

Remarks

None

DRV_SDCARD_POWER_STATE Macro

Defines an override of the power state of the SD Card driver.

File

drv_sdcard_config_template.h

C

```
#define DRV_SDCARD_POWER_STATE SYS_MODULE_POWER_IDLE_STOP
```

Description

SD Card power state configuration

Defines an override of the power state of the SD Card driver.

Remarks

This feature may not be available in the device or the SD Card module selected.

Building the Library

This section lists the files that are available in the SD Card Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/sdcard`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_sdcard.h`	This file provides the interface definitions of the SD Card driver

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_sdcard.c`	This file contains the core implementation of the SD Card driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The SD Card Driver Library depends on the following modules:

- SPI Driver Library
- Clock System Service Library
- Interrupt System Service Library
- Ports System Service Library
- Timer System Service Library
- Timer Driver Library

Library Interface

a) System Level Functions

	Name	Description
	DRV_SDCARD_Initialize	Initializes the SD Card driver. **Implementation:** Dynamic
	DRV_SDCARD_Deinitialize	Deinitializes the specified instance of the SD Card driver module. **Implementation:** Dynamic
	DRV_SDCARD_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_SDCARD_Status	Provides the current status of the SD Card driver module. **Implementation:** Dynamic
	DRV_SDCARD_Tasks	Maintains the driver's state machine. **Implementation:** Dynamic

b) Client Level Functions

	Name	Description
	DRV_SDCARD_Close	Closes an opened-instance of the SD Card driver. **Implementation:** Dynamic
	DRV_SDCARD_Open	Opens the specified SD Card driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SDCARD_Read	Reads blocks of data from the specified block address of the SD Card.
	DRV_SDCARD_Write	Writes blocks of data starting at the specified address of the SD Card.
	DRV_SDCARD_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

c) Status Functions

	Name	Description
	DRV_SDCARD_IsAttached	Returns the physical attach status of the SD Card.
	DRV_SDCARD_IsWriteProtected	Returns the write protect status of the SDCARD.
	DRV_SDCARD_CommandStatus	Gets the current status of the command.
	DRV_SDCARD_GeometryGet	Returns the geometry of the device.

d) Data Types and Constants

	Name	Description
	DRV_SDCARD_INDEX_0	SD Card driver index definitions
	DRV_SDCARD_INDEX_COUNT	Number of valid SD Card driver indices
	DRV_SDCARD_INIT	Contains all the data necessary to initialize the SD Card device
	_DRV_SDCARD_INIT	Contains all the data necessary to initialize the SD Card device
	SDCARD_DETECTION_LOGIC	Defines the different system events
	SDCARD_MAX_LIMIT	Maximum allowed SD card instances
	DRV_SDCARD_INDEX_1	This is macro DRV_SDCARD_INDEX_1.
	DRV_SDCARD_INDEX_2	This is macro DRV_SDCARD_INDEX_2.
	DRV_SDCARD_INDEX_3	This is macro DRV_SDCARD_INDEX_3.
	DRV_SDCARD_COMMAND_HANDLE_INVALID	SDCARD Driver's Invalid Command Handle.
	DRV_SDCARD_COMMAND_HANDLE	Handle identifying commands queued in the driver.

	DRV_SDCARD_COMMAND_STATUS	Identifies the possible events that can result from a request.
	DRV_SDCARD_EVENT	Identifies the possible events that can result from a request.
	DRV_SDCARD_EVENT_HANDLER	Pointer to a SDCARDDriver Event handler function

Description

This section describes the Application Programming Interface (API) functions of the SD Card Driver.

Refer to each section for a detailed description.

a) System Level Functions

DRV_SDCARD_Initialize Function

Initializes the SD Card driver.

Implementation: Dynamic

File

drv_sdcard.h

C

```
SYS_MODULE_OBJ DRV_SDCARD_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT *const
init);
```

Returns

If successful, returns a valid handle to a driver object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the SD Card driver, making it ready for clients to open and use the driver.

Remarks

This routine must be called before any other SD Card routine is called.

This routine should only be called once during system initialization unless DRV_SDCARD_Deinitialize is called to deinitialize the driver instance.

This routine will NEVER block for hardware access. If the operation requires time to allow the hardware to reinitialize, it will be reported by the DRV_SDCARD_Status operation. The system must use DRV_SDCARD_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this routine.

Preconditions

None.

Example

```
DRV_SDCARD_INIT      init;
SYS_MODULE_OBJ       objectHandle;

// Populate the SD Card initialization structure

objectHandle = DRV_SDCARD_Initialize(DRV_SDCARD_INDEX_0, (SYS_MODULE_INIT*)&init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Index for the driver instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the driver. This pointer may be null if no data is required because static overrides have been provided.

Function

SYS_MODULE_OBJ DRV_SDCARD_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

);

DRV_SDCARD_Deinitialize Function

Deinitializes the specified instance of the SD Card driver module.

Implementation: Dynamic

File

drv_sdcard.h

C

```
void DRV_SDCARD_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SD Card driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

This routine will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_SDCARD_Status operation. The system has to use DRV_SDCARD_Status to check if the de-initialization is complete.

Preconditions

Function DRV_SDCARD_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      objectHandle;     //  Returned from DRV_SDCARD_Initialize
SYS_STATUS          status;

DRV_SDCARD_Deinitialize(objectHandle);

status = DRV_SDCARD_Status(objectHandle);
if (SYS_MODULE_UNINITIALIZED == status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SDCARD_Initialize routine.

Function

void DRV_SDCARD_Deinitialize

(

SYS_MODULE_OBJ object

);

DRV_SDCARD_Reinitialize Function

Reinitializes the driver and refreshes any associated hardware settings.

Implementation: Dynamic

File

drv_sdcard.h

C

```
void DRV_SDCARD_Reinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

None

Description

This routine reinitializes the driver and refreshes any associated hardware settings using the given initialization data, but it will not interrupt any ongoing operations.

Remarks

This function can be called multiple times to reinitialize the module.

This operation can be used to refresh any supported hardware registers as specified by the initialization data or to change the power state of the module.

This routine will NEVER block for hardware access. If the operation requires time to allow the hardware to reinitialize, it will be reported by the DRV_SDCARD_Status operation. The system must use DRV_SDCARD_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this routine.

Preconditions

Function DRV_SDCARD_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
DRV_SDCARD_INIT     init;
SYS_MODULE_OBJ      objectHandle;  //  Returned from DRV_SDCARD_Initialize

// Update the required fields of the SD Card initialization structure

DRV_SDCARD_Reinitialize (objectHandle, (SYS_MODULE_INIT*)&init);
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SDCARD_Initialize routine
init	Pointer to the initialization data structure

Function

void DRV_SDCARD_Reinitialize

(

SYS_MODULE_OBJ object,

const SYS_MODULE_INIT * const init

);

DRV_SDCARD_Status Function

Provides the current status of the SD Card driver module.

Implementation: Dynamic

File

drv_sdcard.h

C

```
SYS_STATUS DRV_SDCARD_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another

Note Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_STATUS_BUSY - Indicates that the driver is busy with a previous system level operation and cannot start another

SYS_STATUS_ERROR - Indicates that the driver is in an error state

Description

This routine provides the current status of the SD Card driver module.

Remarks

Any value less than SYS_STATUS_ERROR is also an error state.

SYS_MODULE_DEINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR

This operation can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, then a previous operation has not yet completed. If the status operation returns SYS_STATUS_READY, then it indicates that all previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This routine will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

Function DRV_SDCARD_Initialize must have been called before calling this

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SDCARD_Initialize
SYS_STATUS          status;

status = DRV_SDCARD_Status(object);

if (SYS_MODULE_UNINITIALIZED == status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
else if (SYS_STATUS_ERROR >= status)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SDCARD_Initialize routine

Function

```
SYS_STATUS DRV_SDCARD_Status
(
SYS_MODULE_OBJ object
);
```

DRV_SDCARD_Tasks Function

Maintains the driver's state machine.

Implementation: Dynamic

File

drv_sdcard.h

C

```
void DRV_SDCARD_Tasks(SYS_MODULE_OBJ object);
```

Returns

None

Description

This routine is used to maintain the driver's internal state machine.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_SDCARD_Initialize routine must have been called for the specified SDCARD driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SDCARD_Initialize

while (true)
{
    DRV_SDCARD_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SDCARD_Initialize)

Function

void DRV_SDCARD_Tasks

(

SYS_MODULE_OBJ object

);

b) Client Level Functions

DRV_SDCARD_Close Function

Closes an opened-instance of the SD Card driver.

Implementation: Dynamic

File

drv_sdcard.h

C

```
void DRV_SDCARD_Close(DRV_HANDLE handle);
```

Returns

None

Description

This routine closes an opened-instance of the SD Card driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SDCARD_Open before the caller may use the driver again.

If DRV_IO_INTENT_BLOCKING was requested and the driver was built appropriately to support blocking behavior call may block until the operation is complete.

If DRV_IO_INTENT_NON_BLOCKING request the driver client can call the DRV_SDCARD_Status operation to find out when the module is in the ready state (the handle is no longer valid).

Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SDCARD_Initialize routine must have been called for the specified SD Card driver instance.

DRV_SDCARD_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SDCARD_Open

DRV_SDCARD_Close (handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_SDCARD_Close

(

DRV_HANDLE handle

);

DRV_SDCARD_Open Function

Opens the specified SD Card driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_sdcard.h

C

```
DRV_HANDLE DRV_SDCARD_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID.

Description

This routine opens the specified SD Card driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_SDCARD_Close routine is called.

This routine will NEVER block waiting for hardware.

If the DRV_IO_INTENT_BLOCKING is requested and the driver was built appropriately to support blocking behavior, then other client-level operations may block waiting on hardware until they are complete.

If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID.

Preconditions

Function DRV_SDCARD_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_SDCARD_Open (DRV_SDCARD_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);

if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SDCARD_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT intent

);

DRV_SDCARD_Read Function

Reads blocks of data from the specified block address of the SD Card.

File

drv_sdcard.h

C

```
void DRV_SDCARD_Read(DRV_HANDLE handle, DRV_SDCARD_COMMAND_HANDLE * commandHandle, void *
targetBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_SDCARD_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation for reading blocks of data from the SD Card. The function returns with a valid buffer handle in the commandHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SDCARD_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the driver handle is invalid
- if the target buffer pointer is NULL
- if the number of blocks to be read is zero or more than the actual number of blocks available
- if a buffer object could not be allocated to the request
- if the client opened the driver in write only mode

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SDCARD_EVENT_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SDCARD_EVENT_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

None.

Preconditions

The DRV_SDCARD_Initialize routine must have been called for the specified SDCARD driver instance.

DRV_SDCARD_Open must have been called with DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE as the ioIntent to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = 0x00;
uint32_t    nBlock = 2;
DRV_SDCARD_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySDCARDHandle is the handle returned
// by the DRV_SDCARD_Open function.

DRV_SDCARD_Read(mySDCARDHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SDCARD_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
else
```

```
{
    // Read Successful
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
targetBuffer	Buffer into which the data read from the SD Card will be placed
blockStart	Start block address of the SD Card from where the read should begin.
nBlock	Total number of blocks to be read.

Function

```
void DRV_SDCARD_Read
(
const      DRV_HANDLE handle,
    DRV_SDCARD_COMMAND_HANDLE * commandHandle,
void * targetBuffer,
uint32_t blockStart,
uint32_t nBlock
);
```

DRV_SDCARD_Write Function

Writes blocks of data starting at the specified address of the SD Card.

File

drv_sdcard.h

C

```
void DRV_SDCARD_Write(DRV_HANDLE handle, DRV_SDCARD_COMMAND_HANDLE * commandHandle, void *
sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_SDCARD_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data to the SD Card. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SDCARD_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the source buffer pointer is NULL
- if the client opened the driver for read only
- if the number of blocks to be written is either zero or more than the number of blocks actually available
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SDCARD_EVENT_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SDCARD_EVENT_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

None.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called for the specified SDCARD driver instance.

DRV_SDCARD_Open() routine must have been called to obtain a valid opened device handle. DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified as a parameter to this routine.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = 0x00;
uint32_t    nBlock = 2;
DRV_SDCARD_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySDCARDHandle is the handle returned
// by the DRV_SDCARD_Open function.

// Client registers an event handler with driver

DRV_SDCARD_EventHandlerSet(mySDCARDHandle, APP_SDCARDEventHandler, (uintptr_t)&myAppObj);

DRV_SDCARD_Write(mySDCARDHandle, &commandHandle, &myBuffer, blockStart, nBlock);
```

```
if(DRV_SDCARD_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_SDCARDEventHandler(DRV_SDCARD_EVENT event,
        DRV_SDCARD_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SDCARD_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SDCARD_EVENT_COMMAND_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
sourceBuffer	The source buffer containing data to be programmed to the SD Card.
blockStart	Start block address of SD Card where the writes should begin.
nBlock	Total number of blocks to be written.

Function

void DRV_SDCARD_Write

(

const DRV_HANDLE handle,

DRV_SDCARD_COMMAND_HANDLE * commandHandle,

void * sourceBuffer,

uint32_t blockStart,

uint32_t nBlock

);

DRV_SDCARD_EventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

File

drv_sdcard.h

C

```
void DRV_SDCARD_EventHandlerSet(const DRV_HANDLE handle, const void * eventHandler, const
uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client queues a request for a read or a write operation, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any read or write operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called for the specified SDCARD driver instance.

The DRV_SDCARD_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_SDCARD_COMMAND_HANDLE commandHandle;

// drvSDCARDHandle is the handle returned
// by the DRV_SDCARD_Open function.

// Client registers an event handler with driver. This is done once.

DRV_SDCARD_EventHandlerSet(drvSDCARDHandle, APP_SDCARDEventHandler, (uintptr_t)&myAppObj);

DRV_SDCARD_Read(drvSDCARDHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SDCARD_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_SDCARDEventHandler(DRV_SDCARD_EVENT event,
        DRV_SDCARD_COMMAND_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;
```

```
    switch(event)
    {
        case DRV_SDCARD_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SDCARD_EVENT_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_SDCARD_EventHandlerSet

(

const DRV_HANDLE handle,

const void * eventHandler,

const uintptr_t context

);

c) Status Functions

DRV_SDCARD_IsAttached Function

Returns the physical attach status of the SD Card.

File

drv_sdcard.h

C

```
bool DRV_SDCARD_IsAttached(const DRV_HANDLE handle);
```

Returns

Returns false if the handle is invalid otherwise returns the attach status of the SD Card. Returns true if the SD Card is attached and initialized by the SDCARD driver otherwise returns false.

Description

This function returns the physical attach status of the SD Card.

Remarks

None.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called for the specified SDCARD driver instance.

The DRV_SDCARD_Open() routine must have been called to obtain a valid opened device handle.

Example

```
bool isSDCARDAttached;
isSDCARDAttached = DRV_SDCARD_isAttached(drvSDCARDHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SDCARD_IsAttached

(

const DRV_HANDLE handle

);

DRV_SDCARD_IsWriteProtected Function

Returns the write protect status of the SDCARD.

File

drv_sdcard.h

C

```
bool DRV_SDCARD_IsWriteProtected(const DRV_HANDLE handle);
```

Returns

Returns true if the attached SD Card is write protected. Returns false if the handle is not valid, or if the SD Card is not write protected.

Description

This function returns the physical attach status of the SDCARD. This function returns true if the SD Card is write protected otherwise it returns false.

Remarks

None.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called for the specified SDCARD driver instance.

The DRV_SDCARD_Open() routine must have been called to obtain a valid opened device handle.

Example

```
bool isWriteProtected;
isWriteProtected = DRV_SDCARD_IsWriteProtected(drvSDCARDHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SDCARD_IsWriteProtected

(

const DRV_HANDLE handle

);

DRV_SDCARD_CommandStatus Function

Gets the current status of the command.

File

drv_sdcard.h

C

```
DRV_SDCARD_COMMAND_STATUS DRV_SDCARD_CommandStatus(const DRV_HANDLE handle, const
DRV_SDCARD_COMMAND_HANDLE commandHandle);
```

Returns

A DRV_SDCARD_COMMAND_STATUS value describing the current status of the command. Returns DRV_SDCARD_COMMAND_HANDLE_INVALID if the client handle or the command handle is not valid.

Description

This routine gets the current status of the command. The application must use this routine where the status of a scheduled command needs to be polled on. The function may return DRV_SDCARD_COMMAND_HANDLE_INVALID in a case where the command handle has expired. A command handle expires when the internal buffer object is re-assigned to another read or write request. It is recommended that this function be called regularly in order to track the command status correctly.

The application can alternatively register an event handler to receive read or write operation completion events.

Remarks

This routine will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called.

The DRV_SDCARD_Open() must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE                  handle;         // Returned from DRV_SDCARD_Open
DRV_SDCARD_COMMAND_HANDLE   commandHandle;
DRV_SDCARD_COMMAND_STATUS   status;

status = DRV_SDCARD_CommandStatus(handle, commandHandle);
if(status == DRV_SDCARD_COMMAND_COMPLETED)
{
    // Operation Done
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_SDCARD_COMMAND_STATUS DRV_SDCARD_CommandStatus

(

const DRV_HANDLE handle,

const DRV_SDCARD_COMMAND_HANDLE commandHandle

);

© 2013-2015 Microchip Technology Inc.

DRV_SDCARD_GeometryGet Function

Returns the geometry of the device.

File

drv_sdcard.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_SDCARD_GeometryGet(const DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Pointer to structure which holds the media geometry information.

Description

This API gives the following geometrical details of the SD Card.

- Media Property
- Number of Read/Write/Erase regions in the SD Card
- Number of Blocks and their size in each region of the device

Remarks

None.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called for the specified SDCARD driver instance.

The DRV_SDCARD_Open() routine must have been called to obtain a valid opened device handle.

Example

```
SYS_FS_MEDIA_GEOMETRY * SDCARDGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalSize;

SDCARDGeometry = DRV_SDCARD_GeometryGet(SDCARDOpenHandle1);

readBlockSize  = SDCARDGeometry->geometryTable->blockSize;
nReadBlocks = SDCARDGeometry->geometryTable->numBlocks;
nReadRegions = SDCARDGeometry->numReadRegions;

writeBlockSize  = (SDCARDGeometry->geometryTable +1)->blockSize;
eraseBlockSize  = (SDCARDGeometry->geometryTable +2)->blockSize;

totalSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY * DRV_SDCARD_GeometryGet

(

const DRV_HANDLE handle

);

d) Data Types and Constants

DRV_SDCARD_INDEX_0 Macro

SD Card driver index definitions

File

drv_sdcard.h

C

```
#define DRV_SDCARD_INDEX_0 0
```

Description

SD Card Driver Module Index Numbers

These constants provide SD Card driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_SDCARD_Initialize and DRV_SDCARD_Open routines to identify the driver instance in use.

DRV_SDCARD_INDEX_COUNT Macro

Number of valid SD Card driver indices

File

drv_sdcard.h

C

```
#define DRV_SDCARD_INDEX_COUNT DRV_SDCARD_INDEX_MAX
```

Description

SD Card Driver Module Index Count

This constant identifies number of valid SD Card driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from part-specific header files defined as part of the peripheral libraries.

DRV_SDCARD_INIT Structure

Contains all the data necessary to initialize the SD Card device

File

drv_sdcard.h

C

```
typedef struct _DRV_SDCARD_INIT {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX spiIndex;
  SPI_MODULE_ID spiId;
  CLK_BUSES_PERIPHERAL spiClk;
  uint32_t sdcardSpeedHz;
  SDCARD_DETECTION_LOGIC sdCardPinActiveLogic;
  PORTS_CHANNEL cardDetectPort;
  PORTS_BIT_POS cardDetectBitPosition;
  PORTS_CHANNEL writeProtectPort;
  PORTS_BIT_POS writeProtectBitPosition;
  PORTS_CHANNEL chipSelectPort;
  PORTS_BIT_POS chipSelectBitPosition;
} DRV_SDCARD_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX spiIndex;	SPI driver index
SPI_MODULE_ID spiId;	Identifies peripheral (PLIB-level) ID
CLK_BUSES_PERIPHERAL spiClk;	Peripheral clock used by the SPI
uint32_t sdcardSpeedHz;	SD card communication speed
SDCARD_DETECTION_LOGIC sdCardPinActiveLogic;	SD Card Pin Detection Logic
PORTS_CHANNEL cardDetectPort;	Card detect port
PORTS_BIT_POS cardDetectBitPosition;	Card detect pin
PORTS_CHANNEL writeProtectPort;	Write protect port
PORTS_BIT_POS writeProtectBitPosition;	Write protect pin
PORTS_CHANNEL chipSelectPort;	Chip select port
PORTS_BIT_POS chipSelectBitPosition;	Chip select pin

Description

SD Card Device Driver Initialization Data

This structure contains all the data necessary to initialize the SD Card device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_SDCARD_Initialize routine.

SDCARD_DETECTION_LOGIC Enumeration

Defines the different system events

File

drv_sdcard.h

C

```
typedef enum {
  SDCARD_DETECTION_LOGIC_ACTIVE_LOW,
  SDCARD_DETECTION_LOGIC_ACTIVE_HIGH
} SDCARD_DETECTION_LOGIC;
```

Members

Members	Description
SDCARD_DETECTION_LOGIC_ACTIVE_LOW	The media event is SD Card attach
SDCARD_DETECTION_LOGIC_ACTIVE_HIGH	The media event is SD Card detach

Description

System events

This enum defines different system events.

Remarks

None.

SDCARD_MAX_LIMIT Macro

Maximum allowed SD card instances

File

drv_sdcard.h

C

```
#define SDCARD_MAX_LIMIT 2
```

Description

SD Card Driver Maximum allowed limit

This constant identifies number of valid SD Card driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from part-specific header files defined as part of the peripheral libraries.

© 2013-2015 Microchip Technology Inc.

DRV_SDCARD_INDEX_1 Macro

File

drv_sdcard.h

C

```
#define DRV_SDCARD_INDEX_1 1
```

Description

This is macro DRV_SDCARD_INDEX_1.

DRV_SDCARD_INDEX_2 Macro

File

drv_sdcard.h

C

```
#define DRV_SDCARD_INDEX_2 2
```

Description

This is macro DRV_SDCARD_INDEX_2.

DRV_SDCARD_INDEX_3 Macro

File

drv_sdcard.h

C

```
#define DRV_SDCARD_INDEX_3 3
```

Description

This is macro DRV_SDCARD_INDEX_3.

DRV_SDCARD_COMMAND_HANDLE_INVALID Macro

SDCARD Driver's Invalid Command Handle.

File

drv_sdcard.h

C

```
#define DRV_SDCARD_COMMAND_HANDLE_INVALID SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID
```

Description

SDCARD Driver Invalid Command Handle.

This value defines the SDCARD Driver Invalid Command Handle. This value is returned by read or write routines when the command request was not accepted.

Remarks

None.

DRV_SDCARD_COMMAND_HANDLE Type

Handle identifying commands queued in the driver.

File

drv_sdcard.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_SDCARD_COMMAND_HANDLE;
```

Description

SDCARD Driver command handle.

A command handle is returned by a call to the Read or Write functions. This handle allows the application to track the completion of the operation. This command handle is also returned to the client along with the event that has occurred with respect to the command. This allows the application to connect the event to a specific command in case where multiple commands are queued.

The command handle associated with the command request expires when the client has been notified of the completion of the command (after event handler function that notifies the client returns) or after the command has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SDCARD_COMMAND_STATUS Enumeration

Identifies the possible events that can result from a request.

File

drv_sdcard.h

C

```
typedef enum {
  DRV_SDCARD_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED,
  DRV_SDCARD_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED,
  DRV_SDCARD_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS,
  DRV_SDCARD_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN
} DRV_SDCARD_COMMAND_STATUS;
```

Members

Members	Description
DRV_SDCARD_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED	Done OK and ready
DRV_SDCARD_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED	Scheduled but not started
DRV_SDCARD_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_SDCARD_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN	Unknown Command

Description

SDCARD Driver Events

This enumeration identifies the possible events that can result from a read or a write request made by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SDCARD_EventHandlerSet function when a request is completed.

DRV_SDCARD_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_sdcard.h

C

```
typedef enum {
  DRV_SDCARD_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_SDCARD_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR
} DRV_SDCARD_EVENT;
```

Members

Members	Description
DRV_SDCARD_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE	Operation has been completed successfully.
DRV_SDCARD_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR	There was an error during the operation

Description

SDCARD Driver Events

This enumeration identifies the possible events that can result from a read or a write request issued by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SDCARD_EventHandlerSet function when a request is completed.

DRV_SDCARD_EVENT_HANDLER Type

Pointer to a SDCARDDriver Event handler function

File

drv_sdcard.h

C

```
typedef SYS_FS_MEDIA_EVENT_HANDLER DRV_SDCARD_EVENT_HANDLER;
```

Returns

None.

Description

SDCARD Driver Event Handler Function Pointer

This data type defines the required function signature for the SDCARD event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SDCARD_EVENT_COMMAND_COMPLETE, it means that the write or a erase operation was completed successfully.

If the event is DRV_SDCARD_EVENT_COMMAND_ERROR, it means that the scheduled operation was not completed successfully.

The context parameter contains the handle to the client context, provided at the time the event handling function was registered using the DRV_SDCARD_EventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

Example

```
void APP_MySDCARDEventHandler
(
    DRV_SDCARD_EVENT event,
    DRV_SDCARD_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SDCARD_EVENT_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_SDCARD_EVENT_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write requests

context	Value identifying the context of the application that registered the event handling function

Files

Files

Name	Description
drv_sdcard.h	SD Card Device Driver Interface File
drv_sdcard_config_template.h	SD Card driver configuration definitions template

Description

This section lists the source and header files used by the SD Card Driver Library.

drv_sdcard.h

SD Card Device Driver Interface File

Enumerations

	Name	Description
	DRV_SDCARD_COMMAND_STATUS	Identifies the possible events that can result from a request.
	DRV_SDCARD_EVENT	Identifies the possible events that can result from a request.
	SDCARD_DETECTION_LOGIC	Defines the different system events

Functions

	Name	Description
	DRV_SDCARD_Close	Closes an opened-instance of the SD Card driver. **Implementation:** Dynamic
	DRV_SDCARD_CommandStatus	Gets the current status of the command.
	DRV_SDCARD_Deinitialize	Deinitializes the specified instance of the SD Card driver module. **Implementation:** Dynamic
	DRV_SDCARD_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.
	DRV_SDCARD_GeometryGet	Returns the geometry of the device.
	DRV_SDCARD_Initialize	Initializes the SD Card driver. **Implementation:** Dynamic
	DRV_SDCARD_IsAttached	Returns the physical attach status of the SD Card.
	DRV_SDCARD_IsWriteProtected	Returns the write protect status of the SDCARD.
	DRV_SDCARD_Open	Opens the specified SD Card driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SDCARD_Read	Reads blocks of data from the specified block address of the SD Card.
	DRV_SDCARD_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_SDCARD_Status	Provides the current status of the SD Card driver module. **Implementation:** Dynamic
	DRV_SDCARD_Tasks	Maintains the driver's state machine. **Implementation:** Dynamic
	DRV_SDCARD_Write	Writes blocks of data starting at the specified address of the SD Card.

© 2013-2015 Microchip Technology Inc.

Macros

	Name	Description
	DRV_SDCARD_COMMAND_HANDLE_INVALID	SDCARD Driver's Invalid Command Handle.
	DRV_SDCARD_INDEX_0	SD Card driver index definitions
	DRV_SDCARD_INDEX_1	This is macro DRV_SDCARD_INDEX_1.
	DRV_SDCARD_INDEX_2	This is macro DRV_SDCARD_INDEX_2.
	DRV_SDCARD_INDEX_3	This is macro DRV_SDCARD_INDEX_3.
	DRV_SDCARD_INDEX_COUNT	Number of valid SD Card driver indices
	SDCARD_MAX_LIMIT	Maximum allowed SD card instances

Structures

	Name	Description
	_DRV_SDCARD_INIT	Contains all the data necessary to initialize the SD Card device
	DRV_SDCARD_INIT	Contains all the data necessary to initialize the SD Card device

Types

	Name	Description
	DRV_SDCARD_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_SDCARD_EVENT_HANDLER	Pointer to a SDCARDDriver Event handler function

Description

SD Card Device Driver Interface

The SD Card device driver provides a simple interface to manage the "SD Card" peripheral. This file defines the interface definitions and prototypes for the SD Card driver.

File Name

drv_sdcard.h

Company

Microchip Technology Inc.

drv_sdcard_config_template.h

SD Card driver configuration definitions template

Macros

	Name	Description
	DRV_SDCARD_CLIENTS_NUMBER	Selects the miximum number of clients
	DRV_SDCARD_INDEX_MAX	SD Card Static Index selection
	DRV_SDCARD_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver
	DRV_SDCARD_POWER_STATE	Defines an override of the power state of the SD Card driver.

Description

SD Card Driver Configuration Definitions for the template version

These definitions statically define the driver's mode of operation.

File Name

drv_sdcard_config_template.h

Company

Microchip Technology Inc.

SPI Driver Library

This topic describes the Serial Peripheral Interface (SPI) Driver Library.

Introduction

This library provides an interface to manage the Serial Peripheral Interface (SPI) module on the Microchip family of microcontrollers in different modes of operation.

Description

The SPI module is a full duplex synchronous serial interface useful for communicating with other peripherals or microcontrollers in master/slave relationship and it can transfer data over short distances at high speeds. The peripheral devices may be serial EEPROMs, shift registers, display drivers, analog-to-digital converters, etc. The SPI module is compatible with Motorola's SPI and SIOP interfaces.

During data transfer devices can work either in master or in Slave mode. The source of synchronization is the system clock, which is generated by the master. The SPI module allows one or more slave devices to be connected to a single master device via the same bus.

The SPI serial interface consists of four pins, which are further sub-divided into data and control lines:

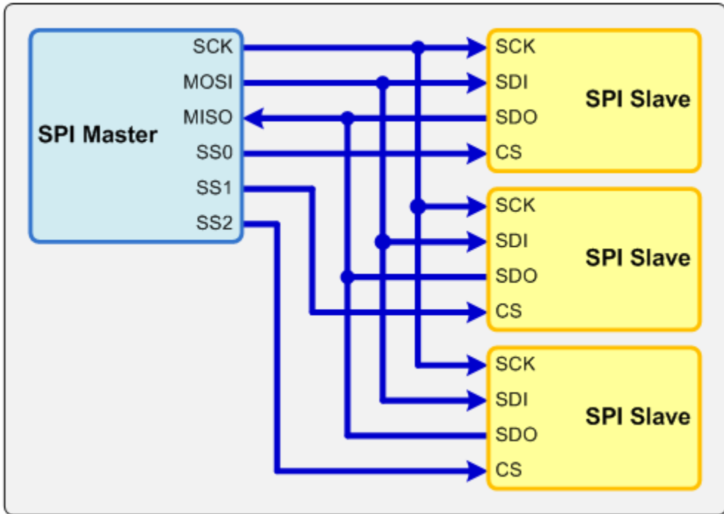
Data Lines:

- MOSI – Master Data Output, Slave Data Input
- MISO – Master Data Input, Slave Data Output

Control Lines:

- SCLK – Serial Clock
- /SS – Slave Select (no addressing)

SPI Master-Slave Relationship



The SPI module can be configured to operate using two, three, or four pins. In the 3-pin mode, the Slave Select line is not used. In the 2-pin mode, both the MOSI and /SS lines are not used.

Note: Third-party trademarks are property of their respective owners. Refer to Software License Agreement for complete licensing information.

Using the Library

This topic describes the basic architecture of the SPI Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_spi.h`

The interface to the SPI Driver library is defined in the `drv_spi.h` header file. Any C language source (`.c`) file that uses the SPI Driver library should include this header.

Please refer to the Understanding MPLAB Harmony section for how the Driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the SPI Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

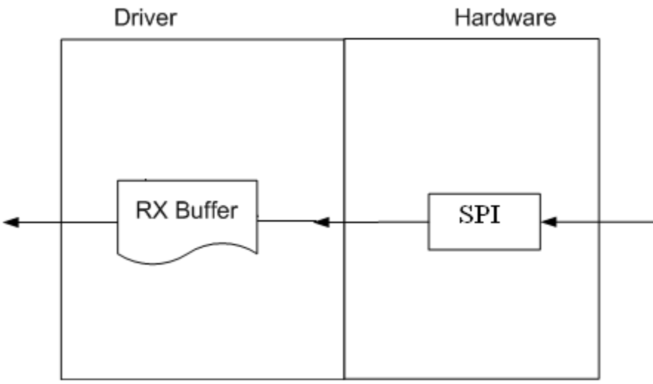
Description

Different types of SPIs are available on Microchip microcontrollers. Some have an internal buffer mechanism and some do not. The buffer depth varies across part families. The SPI driver abstracts out these differences and provides a unified model for data transfer across different types of SPIs available.

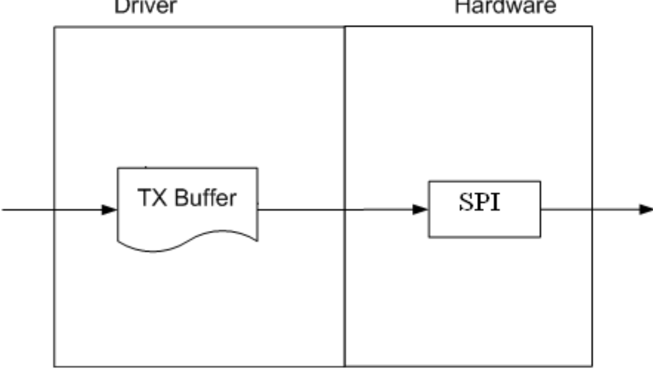
Both transmitter and receiver provides a buffer in the driver which transmits and receives data to/from the hardware. The SPI driver provides a set of interfaces to perform the read and the write.

The following diagrams illustrate the model used by the SPI driver for transmitter and receiver.

Receiver Abstraction Model



Transmitter Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the SPI module.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open, close, status and other setup functions.
Data Transfer Functions	Provides data transfer functions available in the configuration.
Miscellaneous	Provides driver miscellaneous functions, data transfer status function, version identification functions etc.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

Note: Not all modes are available on all devices, please refer to the specific device data sheet to determine the modes that are supported for your device.

System Access

System Initialization and Reinitialization

The system performs the initialization and the reinitialization of the device driver with settings that affect only the instance of the device that is being initialized or reinitialized. During system initialization each instance of the SPI module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_SPI_INIT or by using Initialization Overrides) that are supported by the specific SPI device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section
- The actual peripheral ID enumerated as the PLIB level module ID (e.g., SPI_ID_2)
- Defining the respective interrupt sources for TX, RX, and Error Interrupt

The DRV_SPI_Initialize API returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the Initialize interface would be used by the other system interfaces like DRV_SPI_Deinitialize, DRV_SPI_Status, and DRV_SPI_Tasks.

Note: The system initialization and the reinitialization settings, only affect the instance of the peripheral that is being initialized or reinitialized.

Example:

```
DRV_SPI_INIT                spiInitData;
SYS_MODULE_OBJ              objectHandle;

    spiInitData.moduleInit.value    = SYS_MODULE_POWER_RUN_FULL;
    spiInitData.spiId               = SPI_ID_1;
    spiInitData.taskMode            = DRV_SPI_TASK_MODE_POLLED;
    spiInitData.spiMode             = DRV_SPI_MODE_MASTER;
    spiInitData.spiProtocolType     = DRV_SPI_PROTOCOL_TYPE_STANDARD;
    spiInitData.commWidth           = SPI_COMMUNICATION_WIDTH_8BITS;
    spiInitData.baudRate            = 5000;
    spiInitData.bufferType          = DRV_SPI_BUFFER_TYPE_STANDARD;
                                      // It is highly recommended to set this to
                                      // DRV_SPI_BUFFER_TYPE_ENHANCED for hardware
                                      // that supports it
    spiInitData.inputSamplePhase    = SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE;
    spiInitData.clockMode           = DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_RISE;
    spiInitData.txInterruptSource   = INT_SOURCE_SPI_1_TRANSMIT;
    spiInitData.rxInterruptSource   = INT_SOURCE_SPI_1_RECEIVE;
    spiInitData.errInterruptSource  = INT_SOURCE_SPI_1_ERROR;
    spiInitData.queueSize = 10;
    spiInitData.jobQueueReserveSize = 1;

    objectHandle = DRV_SPI_Initialize(DRV_SPI_INDEX_1, (SYS_MODULE_INIT*)&spiInitData);
    if (SYS_MODULE_OBJ_INVALID == objectHandle)
   {
    // Handle error
  }
```

Tasks Routine

The system will either call DRV_SPI_Tasks, from System Task Service (in a polled environment) or DRV_SPI_Tasks will be called from the ISR of the SPI.

Client Access

General Client Operation

For the application to start using an instance of the module, it must call the DRV_SPI_Open function. This provides the configuration required to open the SPI instance for operation. If the driver is deinitialized using the function DRV_SPI_Deinitialize, the application must call the DRV_SPI_Open function again to set up the instance of the SPI.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

After a client instance is opened, DRV_SPI_ClientConfigure can be called to set a client-specific bps, OperationStarting and OperationEnded callbacks. The OperationStarting callback will be called before the first bit is put onto the SPI bus, allowing for the slave select line to be toggled to active. The OperationEnded callback will be called after the last bit is received, allowing for the slave select line to be toggled to inactive. These two callbacks will be called from the ISR, if the SPI driver is operating in ISR mode, care should be taken that they do the minimum needed. For example, OSAL calls make cause exceptions in ISR context.

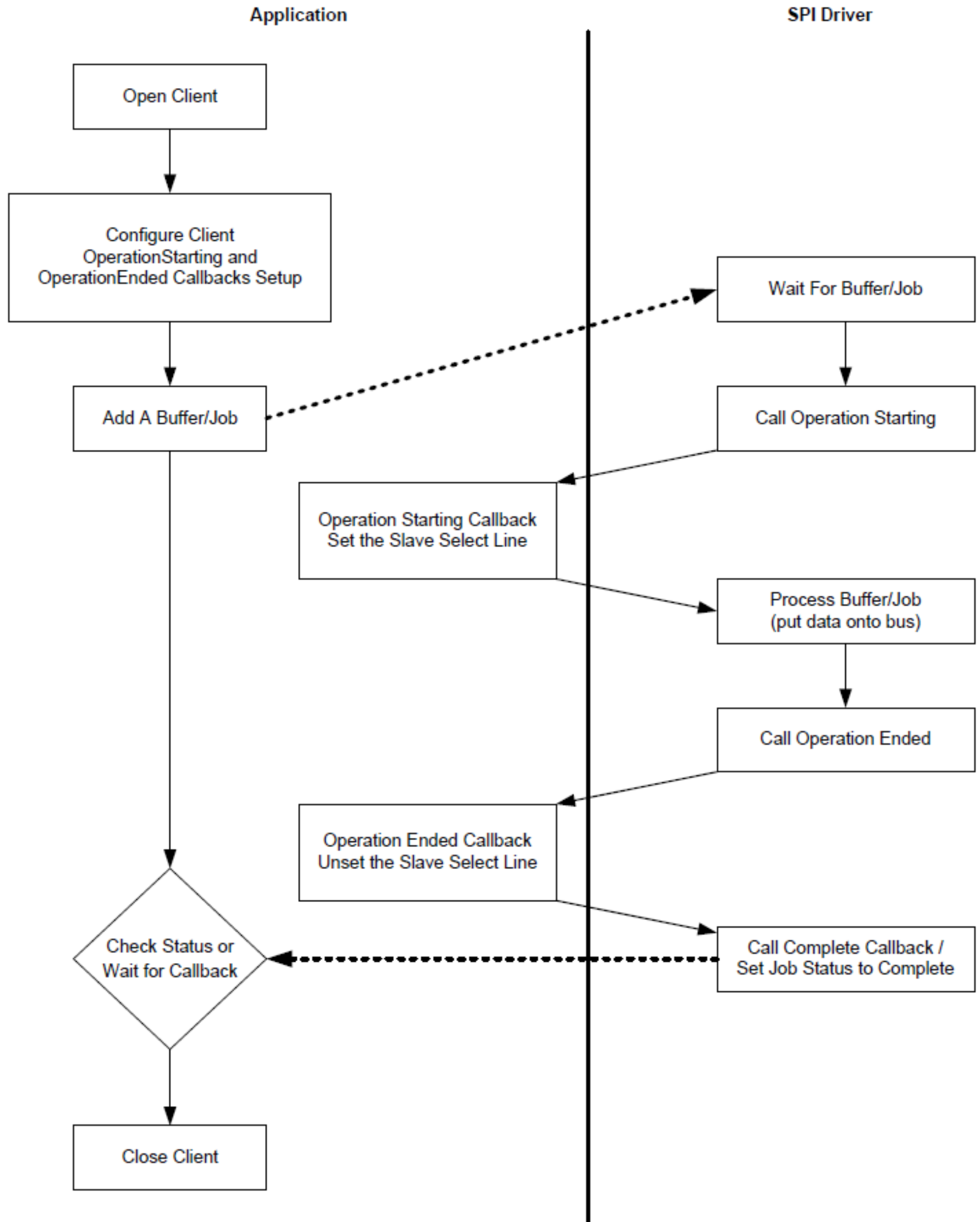
Example:

```
DRV_HANDLE handle;

// Configure the instance DRV_SPI_INDEX_1 with the configuration
handle = DRV_SPI_Open(DRV_SPI_INDEX_1, DRV_IO_INTENT_READWRITE);

if(handle == DRV_HANDLE_INVALID)
{
    // Client cannot open the instance.
}
```

Client Transfer - Core

Client basic functionality provides a extremely basic interface for the driver operation.

The following diagram illustrates the byte/word model used for the data transfer.



Note: It is not necessary to close and reopen the client between multiple transfers.

Client Data Transfer Functionality

Applications using the SPI byte/word functionality, need to perform the following:

1. The system should have completed necessary initialization and the DRV_SPI_Tasks should either be running in polled environment, or in an interrupt environment.
2. Open the driver using DRV_SPI_Open with the necessary intent.
3. Optionally configure the client with DRV_SPI_ClientConfigure to set up OperationStarting and OperationEnded callbacks to handle selecting and deselecting the slave select pin.
4. Add a buffer using the DRV_SPI_BufferAddRead/DRV_SPI_BufferAddWrite/DRV_SPI_BufferAddWriteRead

functions. An optional callback can be provided that will be called when the buffer/job is complete.

5. Check for the current transfer status using DRV_SPI_BufferStatus until the transfer progress is DRV_SPI_BUFFER_EVENT_COMPLETE, or wait for the callback to be called. If the SPI driver is configured in Polled more, ensure that DRV_SPI_Tasks is called regularly to handle the buffer/job.
6. The client will be able to close the driver using DRV_SPI_Close when required.

Example:

```
SYS_MODULE_OBJ spiObject;

int main( void )
{
    while ( 1 )
    {
        appTask ();
        DRV_SPI_Tasks(spiObject);
    }
}
void appTask ()
{
    #define MY_BUFFER_SIZE   5
    DRV_HANDLE          handle;    // Returned from DRV_SPI_Open
    char           myBuffer[MY_BUFFER_SIZE] = { 11, 22, 33, 44, 55};
    unsigned int        numBytes;
    DRV_SPI_BUFFER_HANDLE bufHandle;

    // Preinitialize myBuffer with MY_BUFFER_SIZE bytes of valid data.
    while( 1 )
    {
        switch( state )
        {
            case APP_STATE_INIT:
                /* Initialize the SPI Driver */
                spiObject = DRV_SPI_Initialize( DRV_SPI_INDEX_1,
                                              ( SYS_MODULE_INIT * )
                                                &initConf_1 );

                /* Check for the System Status */
                if( SYS_STATUS_READY != DRV_SPI_Status( spiObject ) )
                    return 0;

                /* Open the Driver */
                handle = DRV_SPI_Open( DRV_SPI_INDEX_1,
                                       DRV_IO_INTENT_EXCLUSIVE );

                /* Enable/Activate the CS */

                /* Update the state to transfer data */
                state = APP_STATE_DATA_PUT;
            break;

            case APP_STATE_DATA_PUT:
                bufHandle = DRV_SPI_BufferAddWrite ( handle, myBuffer,
                                                    5, NULL, NULL );
                /* Update the state to status check */
                state = APP_STATE_DATA_CHECK;
            break;

            case APP_STATE_DATA_CHECK:
                /* Check for the successful data transfer */
                if( DRV_SPI_BUFFER_EVENT_COMPLETE &
                    DRV_SPI_BufferStatus( handle ) )
                {
                    /* Do this repeatedly */
                    state = APP_STATE_DATA_PUT;
                }

            break;
            default:
```

```
            break;
        }
    }
```

Configuring the Library

Miscellaneous Configuration

	Name	Description
	DRV_SPI_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver .
	DRV_SPI_CLIENTS_NUMBER	Selects the maximum number of clients.

System Configuration

	Name	Description
	DRV_SPI_16BIT	Controls the compilation of 16 Bit mode
	DRV_SPI_32BIT	Controls the compilation of 32 Bit mode
	DRV_SPI_8BIT	Controls the compilation of 8 Bit mode
	DRV_SPI_DMA	Controls the compilation of DMA support
	DRV_SPI_DMA_DUMMY_BUFFER_SIZE	Controls the size of DMA dummy buffer
	DRV_SPI_DMA_TXFER_SIZE	Controls the size of DMA transfers
	DRV_SPI_EBM	Controls the compilation of Enhanced Buffer Mode mode
	DRV_SPI_ELEMENTS_PER_QUEUE	Controls the number of elements that are allocated.
	DRV_SPI_ISR	Controls the compilation of ISR mode
	DRV_SPI_MASTER	Controls the compilation of master mode
	DRV_SPI_POLLED	Controls the compilation of Polled mode
	DRV_SPI_RM	Controls the compilation of Standard Buffer mode
	DRV_SPI_SLAVE	Controls the compilation of slave mode

Description

The configuration of the SPI driver is based on the file `system_config.h`.

This header file contains the configuration selection for the SPI driver. Based on the selections made, the SPI driver may support the selected features. These configuration settings will apply to all instances of the SPI driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

System Configuration

DRV_SPI_16BIT Macro

Controls the compilation of 16 Bit mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_16BIT 1
```

Description

SPI 16 Bit Mode Enable

This definition controls whether or not 16 Bit mode functionality is built as part of the driver. With it set to 1 then 16 Bit mode will be compiled and commWidth = SPI_COMMUNICATION_WIDTH_16BITS will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert. With this set the BufferAdd functions will only accept buffer sizes of multiples of 2 (16 bit words)

Remarks

Optional definition

DRV_SPI_32BIT Macro

Controls the compilation of 32 Bit mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_32BIT 1
```

Description

SPI 32 Bit Mode Enable

This definition controls whether or not 32 Bit mode functionality is built as part of the driver. With it set to 1 then 32 Bit mode will be compiled and commWidth = SPI_COMMUNICATION_WIDTH_32BITS will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert. With this set the BufferAdd functions will only accept buffer sizes of multiples of 4 (32 bit words)

Remarks

Optional definition

© 2013-2015 Microchip Technology Inc.

DRV_SPI_8BIT Macro

Controls the compilation of 8 Bit mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_8BIT 1
```

Description

SPI 8 Bit Mode Enable

This definition controls whether or not 8 Bit mode functionality is built as part of the driver. With it set to 1 then 8 Bit mode will be compiled and commWidth = SPI_COMMUNICATION_WIDTH_8BITS will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert.

Remarks

Optional definition

DRV_SPI_DMA Macro

Controls the compilation of DMA support

File

drv_spi_config_template.h

C

```
#define DRV_SPI_DMA 1
```

Description

SPI DMA Enable

This definition controls whether or not DMA functionality is built as part of the driver. With it set to 1 then DMA will be compiled.

Remarks

Optional definition

DRV_SPI_DMA_DUMMY_BUFFER_SIZE Macro

Controls the size of DMA dummy buffer

File

drv_spi_config_template.h

C

```
#define DRV_SPI_DMA_DUMMY_BUFFER_SIZE 256
```

Description

SPI DMA Dummy Buffer Size

This controls the size of the buffer the SPI driver uses to give to the DMA service when it is to send and receive invalid data on the bus. This occurs when the number of bytes to be read are different than the number of bytes transmitted.

Remarks

Optional definition

DRV_SPI_DMA_TXFER_SIZE Macro

Controls the size of DMA transfers

File

drv_spi_config_template.h

C

```
#define DRV_SPI_DMA_TXFER_SIZE 256
```

Description

SPI DMA Transfer Size

This definition controls the maximum number of bytes to transfer per DMA transfer.

Remarks

Optional definition

DRV_SPI_EBM Macro

Controls the compilation of Enhanced Buffer Mode mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_EBM 1
```

Description

SPI Enhanced Buffer Mode Enable (Hardware FIFO)

This definition controls whether or not Enhanced Buffer mode functionality is built as part of the driver. With it set to 1 then enhanced buffer mode will be compiled and bufferType = DRV_SPI_BUFFER_TYPE_ENHANCED will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert. This mode is not available on all PIC32s. Trying to use this mode on PICMX3XX/4XX will cause compile time warnings and errors.

Remarks

Optional definition

DRV_SPI_ELEMENTS_PER_QUEUE Macro

Controls the number of elements that are allocated.

File

drv_spi_config_template.h

C

```
#define DRV_SPI_ELEMENTS_PER_QUEUE 10
```

Description

SPI Buffer Queue Depth

This definition along with DRV_SPI_INSTANCES_NUMBER and DRV_SPI_CLIENT_NUMBER controls how many buffer queue elements are created.

Remarks

Optional definition

DRV_SPI_ISR Macro

Controls the compilation of ISR mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_ISR 1
```

Description

SPI ISR Mode Enable

This definition controls whether or not ISR mode functionality is built as part of the driver. With it set to 1 then ISR mode will be compiled and taskMode = DRV_SPI_TASK_MODE_ISR will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert

Remarks

Optional definition

DRV_SPI_MASTER Macro

Controls the compilation of master mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_MASTER 1
```

Description

SPI Master Mode Enable

This definition controls whether or not master mode functionality is built as part of the driver. With it set to 1 then master mode will be compiled and spiMode = DRV_SPI_MODE_MASTER will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert

Remarks

Optional definition

DRV_SPI_POLLED Macro

Controls the compilation of Polled mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_POLLED 1
```

Description

SPI Polled Mode Enable

This definition controls whether or not polled mode functionality is built as part of the driver. With it set to 1 then polled mode will be compiled and taskMode = DRV_SPI_TASK_MODE_POLLED will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert

Remarks

Optional definition

DRV_SPI_RM Macro

Controls the compilation of Standard Buffer mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_RM 1
```

Description

SPI Standard Buffer Mode Enable

This definition controls whether or not Standard Buffer mode functionality is built as part of the driver. With it set to 1 then standard buffer mode will be compiled and bufferType = DRV_SPI_BUFFER_TYPE_STANDARD will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert. This mode is available on all PIC32s

Remarks

Optional definition

DRV_SPI_SLAVE Macro

Controls the compilation of slave mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_SLAVE 1
```

Description

SPI Slave Mode Enable

This definition controls whether or not slave mode functionality is built as part of the driver. With it set to 1 then slave mode will be compiled and spiMode = DRV_SPI_MODE_SLAVE will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert

Remarks

Optional definition

Miscellaneous Configuration

DRV_SPI_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver .

File

drv_spi_config_template.h

C

```
#define DRV_SPI_INSTANCES_NUMBER 1
```

Description

SPI hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver.

Remarks

Mandatory definition

DRV_SPI_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_spi_config_template.h

C

```
#define DRV_SPI_CLIENTS_NUMBER 1
```

Description

SPI maximum number of clients

This definition selects the maximum number of clients that the SPI driver can support at run time.

Remarks

Mandatory definition

Building the Library

This section lists the files that are available in the SPI Driver Library.

Description

This section list the files that are available in the \src folder of the SPI Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/spi`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_spi.h`	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_spi.c`	Basic SPI Driver implementation file.
`/src/dynamic/drv_spi_api.c`	Functions used by the driver API.
`/src/drv_spi_sys_queue_fifo.c`	Queue implementation used by the SPI Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The SPI Driver Library depends on the following modules:

- Clock System Service Library

Optional Dependencies

- DMA System Service Library (used when operating in DMA mode)
- Interrupt System Service Library (used when task is running in Interrupt mode)

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_SPI_Initialize	Initializes the SPI instance for the specified driver index. **Implementation:** Static/Dynamic
	DRV_SPI_Deinitialize	Deinitializes the specified instance of the SPI driver module. **Implementation:** Static/Dynamic
	DRV_SPI_Status	Provides the current status of the SPI driver module. **Implementation:** Dynamic
	DRV_SPI_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic

b) Client Setup Functions

	Name	Description
	DRV_SPI_Close	Closes an opened instance of the SPI driver. **Implementation:** Dynamic
	DRV_SPI_Open	Opens the specified SPI driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SPI_ClientConfigure	Configures a SPI client with specific data. **Implementation:** Dynamic

c) Data Transfer Functions

	Name	Description
	DRV_SPI_BufferStatus	Returns the transmitter and receiver transfer status. **Implementation:** Dynamic
	DRV_SPI_BufferAddRead	Registers a buffer for a read operation. Actual transfer will happen in the Task function. **Implementation:** Dynamic
	DRV_SPI_BufferAddWrite	Registers a buffer for a write operation. Actual transfer will happen in the Task function. **Implementation:** Dynamic
	DRV_SPI_BufferAddWriteRead	Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddRead2	Registers a buffer for a read operation. Actual transfer will happen in the Task function. **Implementation:** Dynamic
	DRV_SPI_BufferAddWrite2	Registers a buffer for a write operation. Actual transfer will happen in the Task function. **Implementation:** Dynamic
	DRV_SPI_BufferAddWriteRead2	Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPIn_ReceiverBufferIsFull	Returns the receive buffer status. 'n' represents the instance of the SPI driver used. **Implementation:** Static
	DRV_SPIn_TransmitterBufferIsFull	Returns the transmit buffer status. 'n' represents the instance of the SPI driver used. **Implementation:** Static

e) Data Types and Constants

	Name	Description
	DRV_SPI_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_SPI_INDEX_0	SPI driver index definitions.
	DRV_SPI_INDEX_COUNT	Number of valid SPI driver indices.
	DRV_SPI_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_SPI_BUFFER_EVENT_HANDLER	Pointer to a SPI Driver Buffer Event handler function
	DRV_SPI_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the driver.
	DRV_SPI_BUFFER_TYPE	Identifies the various buffer types of the SPI module.
	DRV_SPI_CLOCK_MODE	Identifies the various clock modes of the SPI module.
	DRV_SPI_INIT	Defines the data required to initialize or reinitialize the SPI driver
	_DRV_SPI_INIT	Defines the data required to initialize or reinitialize the SPI driver
	DRV_SPI_MODE	Identifies the various usage modes of the SPI module.
	DRV_SPI_PROTOCOL_TYPE	Identifies the various protocols of the SPI module.
	DRV_SPI_INDEX_1	This is macro DRV_SPI_INDEX_1.
	DRV_SPI_INDEX_2	This is macro DRV_SPI_INDEX_2.
	DRV_SPI_INDEX_3	This is macro DRV_SPI_INDEX_3.
	DRV_SPI_INDEX_4	This is macro DRV_SPI_INDEX_4.
	DRV_SPI_INDEX_5	This is macro DRV_SPI_INDEX_5.
	DRV_SPI_TASK_MODE	Identifies the various modes of how the tasks function will be run.
	DRV_SPI_CLIENT_DATA	Defines the data that can be changed per client.
	_DRV_SPI_CLIENT_DATA	Defines the data that can be changed per client.

Description

This section describes the API functions of the SPI Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_SPI_Initialize Function

Initializes the SPI instance for the specified driver index.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
SYS_MODULE_OBJ DRV_SPI_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const
init);
```

Returns

- If successful - returns a valid handle to a driver instance object
- If unsuccessful - returns SYS_MODULE_OBJ_INVALID

Description

This routine initializes the SPI driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the 'init' parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the SPI module ID. For example, driver instance 0 can be assigned to SPI2. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_SPI_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other SPI routine is called.

This routine should only be called once during system initialization unless DRV_SPI_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
DRV_SPI_INIT        init;
SYS_MODULE_OBJ      objectHandle;

// Populate the SPI initialization structure
init.spiId = SPI_ID_1,
init.taskMode = DRV_SPI_TASK_MODE_ISR,
init.spiMode = DRV_SPI_MODE_MASTER,
init.allowIdleRun = false,
init.spiProtocolType = DRV_SPI_PROTOCOL_TYPE_STANDARD,
init.commWidth = SPI_COMMUNICATION_WIDTH_8BITS,
init.spiClk = CLK_BUS_PERIPHERAL_2,
init.baudRate = 10000000,
init.bufferType = DRV_SPI_BUFFER_TYPE_ENHANCED,
init.clockMode = DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL,
init.inputSamplePhase = SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE,
init.txInterruptSource = INT_SOURCE_SPI_1_TRANSMIT,
init.rxInterruptSource = INT_SOURCE_SPI_1_RECEIVE,
init.errInterruptSource = INT_SOURCE_SPI_1_ERROR,
init.queueSize = 10,
init.jobQueueReserveSize = 1,

objectHandle = DRV_SPI_Initialize(DRV_SPI_INDEX_1, (SYS_MODULE_INIT*)usartInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized. Please note this is not the SPI id. The hardware SPI id is set in the initialization structure. This is the index of the driver index to use.
init	Pointer to a data structure containing any data necessary to initialize the driver. If this pointer is NULL, the driver uses the static initialization override macros for each member of the initialization data structure.

Function

SYS_MODULE_OBJ DRV_SPI_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

DRV_SPI_Deinitialize Function

Deinitializes the specified instance of the SPI driver module.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
void DRV_SPI_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SPI driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_SPI_Status operation. The system has to use DRV_SPI_Status to find out when the module is in the ready state.

Preconditions

Function DRV_SPI_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_SPI_Initialize
SYS_STATUS          status;

DRV_SPI_Deinitialize ( object );

status = DRV_SPI_Status( object );
if( SYS_MODULE_UNINITIALIZED == status )
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_SPI_Initialize

Function

void DRV_SPI_Deinitialize (SYS_MODULE_OBJ object)

DRV_SPI_Status Function

Provides the current status of the SPI driver module.

Implementation: Dynamic

File

drv_spi.h

C

```
SYS_STATUS DRV_SPI_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that the driver is busy with a previous

system level operation and cannot start another

Description

This function provides the current status of the SPI driver module.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_MODULE_UNINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

This function can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_SPI_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SPI_Initialize
SYS_STATUS          status;

status = DRV_SPI_Status( object );
if( SYS_STATUS_READY != status )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_SPI_Initialize

Function

SYS_STATUS DRV_SPI_Status (SYS_MODULE_OBJ object)

DRV_SPI_Tasks Function

Maintains the driver's state machine and implements its ISR.

Implementation: Dynamic

File

drv_spi.h

C

```
void DRV_SPI_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal state machine and implement its transmit ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks() function. In interrupt mode, this function should be called in the transmit interrupt service routine of the USART that is associated with this USART driver hardware instance.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This function may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SPI_Initialize

while( true )
{
    DRV_SPI_Tasks ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SPI_Initialize)

Function

void DRV_SPI_Tasks (SYS_MODULE_OBJ object);

b) Client Setup Functions

DRV_SPI_Close Function

Closes an opened instance of the SPI driver.

Implementation: Dynamic

File

drv_spi.h

C

```
void DRV_SPI_Close(DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened instance of the SPI driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SPI_Open before the caller may use the driver again. This function is thread safe in a RTOS application.

Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SPI_Open

DRV_SPI_Close ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_SPI_Close (DRV_HANDLE handle)

DRV_SPI_Open Function

Opens the specified SPI driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_spi.h

C

```
DRV_HANDLE DRV_SPI_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. An error can occur when the following is true:

- if the number of client objects allocated via DRV_SPI_INSTANCES_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid

Description

This routine opens the specified SPI driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

If ioIntent is DRV_IO_INTENT_READ, the client will only be read from the driver. If ioIntent is DRV_IO_INTENT_WRITE, the client will only be able to write to the driver. If the ioIntent in DRV_IO_INTENT_READWRITE, the client will be able to do both, read and write.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_SPI_Close routine is called. This routine will NEVER block waiting for hardware. If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

The DRV_SPI_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_SPI_Open( DRV_SPI_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );

if( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Index of the driver initialized with DRV_SPI_Initialize(). Please note this is not the SPI ID.
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SPI_Open (const SYS_MODULE_INDEX drvIndex,

```
const DRV_IO_INTENT ioIntent )
```

DRV_SPI_ClientConfigure Function

Configures a SPI client with specific data.

Implementation: Dynamic

File

drv_spi.h

C

```
int32_t DRV_SPI_ClientConfigure(DRV_HANDLE handle, const DRV_SPI_CLIENT_DATA * cfgData);
```

Returns

- If successful - the routing will return greater than or equal to zero
- If an error occurs - the return value is negative

Description

This routine takes a DRV_SPI_CLIENT_DATA structure and sets client specific options. Whenever a new SPI job is started these values will be used. Passing in NULL will reset the client back to configuration parameters passed to driver initialization. A zero in any of the structure elements will reset that specific configuration back to the driver default.

Preconditions

The DRV_SPI_Open function must have been called before calling this function.

Parameters

Parameters	Description
handle	handle of the client returned by DRV_SPI_Open.
cfgData	Client-specific configuration data.

Function

int32_t DRV_SPI_ClientConfigure (DRV_HANDLE handle,

const DRV_SPI_CLIENT_DATA * cfgData)

c) Data Transfer Functions

DRV_SPI_BufferStatus Function

Returns the transmitter and receiver transfer status.

Implementation: Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_EVENT DRV_SPI_BufferStatus(DRV_SPI_BUFFER_HANDLE bufferHandle);
```

Returns

A DRV_SPI_BUFFER_STATUS value describing the current status of the transfer.

Description

This returns the transmitter and receiver transfer status.

Remarks

The returned status may contain a value with more than one of the bits specified in the DRV_SPI_BUFFER_STATUS enumeration set. The caller should perform an AND with the bit of interest and verify if the result is non-zero (as shown in the example) to verify the desired status bit.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;    // Returned from DRV_SPI_Open

if( DRV_SPI_BUFFER_STATUS_SUCCESS & DRV_SPI_BufferStatus( handle ) )
{
    // All transmitter data has been sent.
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_SPI_BUFFER_STATUS DRV_SPI_BufferStatus (DRV_SPI_BUFFER_HANDLE bufferHandle)

DRV_SPI_BufferAddRead Function

Registers a buffer for a read operation. Actual transfer will happen in the Task function.

Implementation: Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddRead(DRV_HANDLE handle, void * rxBuffer, size_t size,
DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void * context);
```

Returns

None.

Description

Registers a buffer for a read operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

If the driver was not able to accept the data, an overrun error status will be captured. To ensure that the driver is ready to accept data, the caller must first check the return value to DRV_SPI_BufferStatus, as shown in the example.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SPI_Open call.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char    myBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle;

switch ( state )
{
    case 0:
        // PUT API returns data in any case, it is up to the user to use it
        bufferHandle = DRV_SPI_BufferAddRead( handle, myBuffer, 10, NULL, NULL );
        state++;
        break;
    case 1:
        if( DRV_SPI_BUFFER_STATUS_SUCCESS & DRV_SPI_BufferStatus( bufferHandle ) )
        {
            state++;
            // All transmitter data has been sent.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
rxBuffer	The buffer to which the data should be written to.
size	Number of bytes to be read from the SPI bus.
completeCB	Pointer to a function to be called when this queued operation is complete
context	unused by the driver but this is passed to the callback when it is called

Function

DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddRead (DRV_HANDLE handle, void *rxBuffer,
size_t size, DRV_SPI_BUFFER_EVENT_HANDLER completeCB,
void * context)

DRV_SPI_BufferAddWrite Function

Registers a buffer for a write operation. Actual transfer will happen in the Task function.

Implementation: Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWrite(DRV_HANDLE handle, void * txBuffer, size_t size,
DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void * context);
```

Returns

None.

Description

Registers a buffer for a write operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

None.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SPI_Open call.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char    myBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle;

switch ( state )
{
    case 0:
        // PUT API returns data in any case, it is up to the user to use it
        bufferHandle = DRV_SPI_BufferAddWrite( handle, myBuffer, 10, NULL, NULL );
        state++;
        break;
    case 1:
        if( DRV_SPI_BUFFER_STATUS_SUCCESS & DRV_SPI_BufferStatus( bufferHandle ) )
        {
            state++;
            // All transmitter data has been sent.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txBuffer	The buffer which hold the data.
size	Number of bytes to be written to the SPI bus.
completeCB	Pointer to a function to be called when this queued operation is complete
context	unused by the driver but this is passed to the callback when it is called

Function

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWrite ( DRV_HANDLE handle, void *txBuffer,
size_t size,                           DRV_SPI_BUFFER_EVENT_HANDLER completeCB,
void * context )
```

DRV_SPI_BufferAddWriteRead Function

Registers a buffer for a read and write operation. Actual transfer will happen in the Task function.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWriteRead(DRV_HANDLE handle, void * txBuffer, size_t
txSize, void * rxBuffer, size_t rxSize, DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void *
context);
```

Returns

None.

Description

Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

If the driver was not able to accept the data, an overrun error status will be captured. To ensure that the driver is ready to accept data, the caller must first check the return value to DRV_SPI_BufferStatus, as shown in the example.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char   myReadBuffer[MY_BUFFER_SIZE], myWriteBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle;

switch ( state )
{
    case 0:
        // PUT API returns data in any case, it is up to the user to use it
        bufferHandle = DRV_SPI_BufferAddWriteRead( handle, myWriteBuffer,
            myReadBuffer, 10, NULL, NULL );

        state++;
        break;
    case 1:
        if( DRV_SPI_BUFFER_STATUS_SUCCESS & DRV_SPI_BufferStatus( bufferHandle ) )
        {
            state++;
            // All transmitter data has been sent.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txBuffer	The buffer which hold the data.
txSize	Number of bytes to be written to the SPI bus.
rxBuffer	The buffer to which the data should be written to.
rxSize	Number of bytes to be read from the SPI bus

completeCB	Pointer to a function to be called when this queued operation is complete
context	unused by the driver but this is passed to the callback when it is called

Function

DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWriteRead(DRV_HANDLE handle,
void *txBuffer, void *rxBuffer, size_t size,)

DRV_SPI_BufferAddRead2 Function

Registers a buffer for a read operation. Actual transfer will happen in the Task function.

Implementation: Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddRead2(DRV_HANDLE handle, void * rxBuffer, size_t size,
DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void * context, DRV_SPI_BUFFER_HANDLE * jobHandle);
```

Returns

None.

Description

Registers a buffer for a read operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

If the driver was not able to accept the data, an overrun error status will be captured. To ensure that the driver is ready to accept data, the caller must first check the return value to DRV_SPI_BufferStatus, as shown in the example.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SPI_Open call.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char    myBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle;

switch ( state )
{
    case 0:
        // PUT API returns data in any case, it is up to the user to use it
        bufferHandle = DRV_SPI_BufferAddRead2( handle, myBuffer, 10, NULL, NULL, NULL );
        state++;
        break;
    case 1:
        if( DRV_SPI_BUFFER_STATUS_SUCCESS & DRV_SPI_BufferStatus( bufferHandle ) )
        {
            state++;
            // All transmitter data has been sent.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
rxBuffer	The buffer to which the data should be written to.
size	Number of bytes to be read from the SPI bus.
completeCB	Pointer to a function to be called when this queued operation is complete
context	unused by the driver but this is passed to the callback when it is called

jobHandle	pointer to the buffer handle, this will be set before the function returns and can be used in the ISR callback.

Function

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddRead2 ( DRV_HANDLE handle, void *rxBuffer,
size_t size,                    DRV_SPI_BUFFER_EVENT_HANDLER completeCB,
void * context,                 DRV_SPI_BUFFER_HANDLE * jobHandle )
```

DRV_SPI_BufferAddWrite2 Function

Registers a buffer for a write operation. Actual transfer will happen in the Task function.

Implementation: Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWrite2(DRV_HANDLE handle, void * txBuffer, size_t size,
DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void * context, DRV_SPI_BUFFER_HANDLE * jobHandle);
```

Returns

None.

Description

Registers a buffer for a write operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

None.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SPI_Open call.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char   myBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle;

switch ( state )
{
    case 0:
        // PUT API returns data in any case, it is up to the user to use it
        bufferHandle = DRV_SPI_BufferAddWrite2( handle, myBuffer, 10, NULL, NULL, NULL );
        state++;
        break;
    case 1:
        if( DRV_SPI_BUFFER_STATUS_SUCCESS & DRV_SPI_BufferStatus( bufferHandle ) )
        {
            state++;
            // All transmitter data has been sent.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txBuffer	The buffer which hold the data.
size	Number of bytes to be written to the SPI bus.
completeCB	Pointer to a function to be called when this queued operation is complete
context	unused by the driver but this is passed to the callback when it is called
jobHandle	pointer to the buffer handle, this will be set before the function returns and can be used in the ISR callback.

Function

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWrite ( DRV_HANDLE handle, void *txBuffer,
size_t size,                          DRV_SPI_BUFFER_EVENT_HANDLER completeCB,
void * context,                             DRV_SPI_BUFFER_HANDLE * jobHandle )
```

DRV_SPI_BufferAddWriteRead2 Function

Registers a buffer for a read and write operation. Actual transfer will happen in the Task function.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWriteRead2(DRV_HANDLE handle, void * txBuffer, size_t
txSize, void * rxBuffer, size_t rxSize, DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void *
context, DRV_SPI_BUFFER_HANDLE * jobHandle);
```

Returns

None.

Description

Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

If the driver was not able to accept the data, an overrun error status will be captured. To ensure that the driver is ready to accept data, the caller must first check the return value to DRV_SPI_BufferStatus, as shown in the example.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char   myReadBuffer[MY_BUFFER_SIZE], myWriteBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle;

switch ( state )
{
    case 0:
        // PUT API returns data in any case, it is up to the user to use it
        bufferHandle = DRV_SPI_BufferAddWriteRead2( handle, myWriteBuffer,
            myReadBuffer, 10, NULL, NULL, NULL );

        state++;
        break;
    case 1:
        if( DRV_SPI_BUFFER_STATUS_SUCCESS & DRV_SPI_BufferStatus( bufferHandle ) )
        {
            state++;
            // All transmitter data has been sent.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txBuffer	The buffer which hold the data.
txSize	Number of bytes to be written to the SPI bus.
rxBuffer	The buffer to which the data should be written to.
rxSize	Number of bytes to be read from the SPI bus

completeCB	Pointer to a function to be called when this queued operation is complete
context	unused by the driver but this is passed to the callback when it is called
jobHandle	pointer to the buffer handle, this will be set before the function returns and can be used in the ISR callback.

Function

DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWriteRead(DRV_HANDLE handle,
void *txBuffer, void *rxBuffer, size_t size,
DRV_SPI_BUFFER_EVENT_HANDLER completeCB,
void * context, DRV_SPI_BUFFER_HANDLE * jobHandle)

DRV_SPIn_ReceiverBufferIsFull Function

Returns the receive buffer status. 'n' represents the instance of the SPI driver used.

Implementation: Static

File

drv_spi.h

C

```
bool DRV_SPIn_ReceiverBufferIsFull();
```

Returns

Receive Buffer Status

- 1 - Full
- 0 - Empty

Description

This function returns the receive buffer status (full/empty).

Remarks

None.

Preconditions

None.

Example

```
bool rxBufStat;

rxBufStat = DRV_SPIn_ReceiverBufferIsFull();

if (rxBufStat)
{
...
}
```

Function

bool DRV_SPIn_ReceiverBufferIsFull(void)

© 2013-2015 Microchip Technology Inc.

DRV_SPIn_TransmitterBufferIsFull Function

Returns the transmit buffer status. 'n' represents the instance of the SPI driver used.

Implementation: Static

File

drv_spi.h

C

```
bool DRV_SPIn_TransmitterBufferIsFull();
```

Returns

Transmit Buffer Status

- 1 - Full
- 0 - Empty

Description

This function returns the transmit buffer status (full/empty).

Remarks

None.

Preconditions

None.

Example

```
bool txBufStat;

txBufStat = DRV_SPIn_TransmitterBufferIsFull();

if (txBufStat)
{
...
}
```

Function

bool DRV_SPIn_TransmitterBufferIsFull(void)

d) Miscellaneous Functions

e) Data Types and Constants

DRV_SPI_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_spi.h

C

```
#define DRV_SPI_BUFFER_HANDLE_INVALID ((DRV_SPI_BUFFER_HANDLE)(-1))
```

Description

SPI Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_SPI_BufferAddRead() and DRV_SPI_BufferAddWrite() function if the buffer add request was not successful.

Remarks

None.

DRV_SPI_INDEX_0 Macro

SPI driver index definitions.

File

drv_spi.h

C

```
#define DRV_SPI_INDEX_0 0
```

Description

SPI Driver Module Index Numbers

These constants provide the SPI driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_SPI_Initialize and DRV_SPI_Open functions to identify the driver instance in use.

DRV_SPI_INDEX_COUNT Macro

Number of valid SPI driver indices.

File

drv_spi.h

C

```
#define DRV_SPI_INDEX_COUNT SPI_NUMBER_OF_MODULES
```

Description

SPI Driver Module Index Count

This constant identifies the number of valid SPI driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific header files defined as part of the peripheral libraries.

DRV_SPI_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_spi.h

C

```
typedef enum {
  DRV_SPI_BUFFER_EVENT_PENDING,
  DRV_SPI_BUFFER_EVENT_PROCESSING,
  DRV_SPI_BUFFER_EVENT_COMPLETE,
  DRV_SPI_BUFFER_EVENT_ERROR
} DRV_SPI_BUFFER_EVENT;
```

Members

Members	Description
DRV_SPI_BUFFER_EVENT_PENDING	Buffer is pending to get processed
DRV_SPI_BUFFER_EVENT_PROCESSING	Buffer is being processed
DRV_SPI_BUFFER_EVENT_COMPLETE	All data from or to the buffer was transferred successfully.
DRV_SPI_BUFFER_EVENT_ERROR	There was an error while processing the buffer transfer request.

Description

SPI Driver Buffer Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_SPI_BufferAddRead or DRV_SPI_BufferAddWrite functions.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_SPI_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_SPI_BUFFER_EVENT_HANDLER Type

Pointer to a SPI Driver Buffer Event handler function

File

drv_spi.h

C

```
typedef void (* DRV_SPI_BUFFER_EVENT_HANDLER)(DRV_SPI_BUFFER_EVENT event, DRV_SPI_BUFFER_HANDLE
bufferHandle, void * context);
```

Returns

None.

Description

SPI Driver Buffer Event Handler Function Pointer

This data type defines the required function signature for the SPI driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values and return value are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SPI_BUFFER_EVENT_COMPLETE, it means that the data was transferred successfully.

If the event is DRV_SPI_BUFFER_EVENT_ERROR, it means that the data was not transferred successfully.

The bufferHandle parameter contains the buffer handle of the buffer that failed.

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_SPI_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in an interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive operations with in this function.

Example

```
void APP_MyBufferEventHandler( DRV_SPI_BUFFER_EVENT event,
                               DRV_SPI_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SPI_BUFFER_EVENT_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_SPI_BUFFER_EVENT_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the vent relates
context	Value identifying the context of the application that registered the event handling function.

DRV_SPI_BUFFER_HANDLE Type

Handle identifying a read or write buffer passed to the driver.

File

drv_spi.h

C

```
typedef uintptr_t DRV_SPI_BUFFER_HANDLE;
```

Description

SPI Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_SPI_BufferAddRead()/ DRV_SPI_BufferAddWrite or DRV_SPI_BufferAddReadWrite() functions. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SPI_BUFFER_TYPE Enumeration

Identifies the various buffer types of the SPI module.

File

drv_spi.h

C

```
typedef enum {
  DRV_SPI_BUFFER_TYPE_STANDARD,
  DRV_SPI_BUFFER_TYPE_ENHANCED
} DRV_SPI_BUFFER_TYPE;
```

Members

Members	Description
DRV_SPI_BUFFER_TYPE_STANDARD	SPI Buffer Type Standard
DRV_SPI_BUFFER_TYPE_ENHANCED	SPI Enhanced Buffer Type

Description

SPI Buffer Type Selection

This enumeration identifies the various buffer types of the SPI module.

Remarks

None.

DRV_SPI_CLOCK_MODE Enumeration

Identifies the various clock modes of the SPI module.

File

drv_spi.h

C

```
typedef enum {
  DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_RISE,
  DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL,
  DRV_SPI_CLOCK_MODE_IDLE_HIGH_EDGE_FALL,
  DRV_SPI_CLOCK_MODE_IDLE_HIGH_EDGE_RISE
} DRV_SPI_CLOCK_MODE;
```

Members

Members	Description
DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_RISE	SPI Clock Mode 0 - Idle State Low & Sampling on Rising Edge
DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL	SPI Clock Mode 1 - Idle State Low & Sampling on Falling Edge
DRV_SPI_CLOCK_MODE_IDLE_HIGH_EDGE_FALL	SPI Clock Mode 2 - Idle State High & Sampling on Falling Edge
DRV_SPI_CLOCK_MODE_IDLE_HIGH_EDGE_RISE	SPI Clock Mode 3 - Idle State High & Sampling on Rising Edge

Description

SPI Clock Mode Selection

This enumeration identifies the various clock modes of the SPI module.

Remarks

None.

DRV_SPI_INIT Structure

Defines the data required to initialize or reinitialize the SPI driver

File

drv_spi.h

C

```
typedef struct _DRV_SPI_INIT {
  SYS_MODULE_INIT moduleInit;
  SPI_MODULE_ID spiId;
  DRV_SPI_TASK_MODE taskMode;
  DRV_SPI_MODE spiMode;
  bool allowIdleRun;
  DRV_SPI_PROTOCOL_TYPE spiProtocolType;
  SPI_FRAME_SYNC_PULSE frameSyncPulse;
  SPI_FRAME_PULSE_POLARITY framePulsePolarity;
  SPI_FRAME_PULSE_DIRECTION framePulseDirection;
  SPI_FRAME_PULSE_EDGE framePulseEdge;
  SPI_FRAME_PULSE_WIDTH framePulseWidth;
  SPI_AUDIO_TRANSMIT_MODE audioTransmitMode;
  SPI_AUDIO_PROTOCOL audioProtocolMode;
  SPI_COMMUNICATION_WIDTH commWidth;
  CLK_BUSES_PERIPHERAL spiClk;
  uint32_t baudRate;
  DRV_SPI_BUFFER_TYPE bufferType;
  DRV_SPI_CLOCK_MODE clockMode;
  SPI_INPUT_SAMPLING_PHASE inputSamplePhase;
  INT_SOURCE txInterruptSource;
  INT_SOURCE rxInterruptSource;
  INT_SOURCE errInterruptSource;
  uint8_t numTrfsSmPolled;
  DMA_CHANNEL txDmaChannel;
  uint8_t txDmaThreshold;
  DMA_CHANNEL rxDmaChannel;
  uint8_t rxDmaThreshold;
  uint8_t queueSize;
  uint8_t jobQueueReserveSize;
  DRV_SPI_BUFFER_EVENT_HANDLER operationStarting;
  DRV_SPI_BUFFER_EVENT_HANDLER operationEnded;
} DRV_SPI_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SPI_MODULE_ID spiId;	Identifies peripheral (PLIB-level) ID
DRV_SPI_TASK_MODE taskMode;	SPI Task Mode Type
DRV_SPI_MODE spiMode;	SPI Usage Mode Type
bool allowIdleRun;	Allow SPI to run when CPU goes to idle mode
DRV_SPI_PROTOCOL_TYPE spiProtocolType;	SPI Protocol Type
SPI_FRAME_SYNC_PULSE frameSyncPulse;	Framed mode Sync Pulse
SPI_FRAME_PULSE_POLARITY framePulsePolarity;	Framed Mode pulse polarity
SPI_FRAME_PULSE_DIRECTION framePulseDirection;	Framed Mode Pulse Direction
SPI_FRAME_PULSE_EDGE framePulseEdge;	Framed mode Pulse Edge

SPI_FRAME_PULSE_WIDTH framePulseWidth;	Framed Mode Pulse width
SPI_AUDIO_TRANSMIT_MODE audioTransmitMode;	Audio mode transmit mode
SPI_AUDIO_PROTOCOL audioProtocolMode;	Audio mode protocol mode
SPI_COMMUNICATION_WIDTH commWidth;	Communication Width
CLK_BUSES_PERIPHERAL spiClk;	Peripheral clock used by the SPI driver
uint32_t baudRate;	Baud Rate Value
DRV_SPI_BUFFER_TYPE bufferType;	SPI Buffer Type
DRV_SPI_CLOCK_MODE clockMode;	SPI Clock mode
SPI_INPUT_SAMPLING_PHASE inputSamplePhase;	SPI Input Sample Phase Selection
INT_SOURCE txInterruptSource;	Transmit/Receive or Transmit Interrupt Source for SPI module
INT_SOURCE rxInterruptSource;	Receive Interrupt Source for SPI module
INT_SOURCE errInterruptSource;	Error Interrupt Source for SPI module
uint8_t numTrfsSmPolled;	While using standard buffer and polled mode how many transfers to do before yielding to other tasks
DMA_CHANNEL txDmaChannel;	DMA Channel for the Transmitter
uint8_t txDmaThreshold;	Threshold for the minimum number of bytes to send to use DMA
DMA_CHANNEL rxDmaChannel;	DMA Channel for the Receiver
uint8_t rxDmaThreshold;	Threshold for the minimum number of bytes to receive to use DMA
uint8_t queueSize;	This is the buffer queue size. This is the maximum number of transfer requests that driver will queue.
uint8_t jobQueueReserveSize;	This controls the minimum number of jobs that the driver will be able to accept without running out of memory. The driver will reserve this number of jobs from the global SPI queue so that it will always be available
DRV_SPI_BUFFER_EVENT_HANDLER operationStarting;	This callback is fired when an operation is about to start on the SPI bus. This allows the user to set any pins that need to be set. This callback may be called from an ISR so should not include OSAL calls. The context parameter is the same one passed into the BufferAddRead, BufferAddWrite, BufferAddWriteRead function.
DRV_SPI_BUFFER_EVENT_HANDLER operationEnded;	This callback is fired when an operation has just completed on the SPI bus. This allows the user to set any pins that need to be set. This callback may be called from an ISR so should not include OSAL calls. The context parameter is the same one passed into the BufferAddRead, BufferAddWrite, BufferAddWriteRead function.

Description

SPI Driver Initialization Data

This data type defines the data required to initialize or reinitialize the SPI driver. If the driver is built statically, the members of this data structure are statically over-ridden by static override definitions in the system_config.h file.

Remarks

None.

DRV_SPI_MODE Enumeration

Identifies the various usage modes of the SPI module.

File

drv_spi.h

C

```
typedef enum {
  DRV_SPI_MODE_MASTER,
  DRV_SPI_MODE_SLAVE
} DRV_SPI_MODE;
```

Members

Members	Description
DRV_SPI_MODE_MASTER	SPI Mode Master
DRV_SPI_MODE_SLAVE	SPI Mode Slave

Description

SPI Usage Modes Enumeration

This enumeration identifies the various usage modes of the SPI module.

Remarks

None.

DRV_SPI_PROTOCOL_TYPE Enumeration

Identifies the various protocols of the SPI module.

File

drv_spi.h

C

```
typedef enum {
  DRV_SPI_PROTOCOL_TYPE_STANDARD,
  DRV_SPI_PROTOCOL_TYPE_FRAMED,
  DRV_SPI_PROTOCOL_TYPE_AUDIO
} DRV_SPI_PROTOCOL_TYPE;
```

Members

Members	Description
DRV_SPI_PROTOCOL_TYPE_STANDARD	SPI Protocol Type Standard
DRV_SPI_PROTOCOL_TYPE_FRAMED	SPI Protocol Type Framed
DRV_SPI_PROTOCOL_TYPE_AUDIO	SPI Protocol Type Audio

Description

SPI Protocols Enumeration

This enumeration identifies the various protocols of the SPI module.

Remarks

None.

DRV_SPI_INDEX_1 Macro

File

drv_spi.h

C

```
#define DRV_SPI_INDEX_1 1
```

Description

This is macro DRV_SPI_INDEX_1.

DRV_SPI_INDEX_2 Macro

File

drv_spi.h

C

```
#define DRV_SPI_INDEX_2 2
```

Description

This is macro DRV_SPI_INDEX_2.

© 2013-2015 Microchip Technology Inc.

DRV_SPI_INDEX_3 Macro

File

drv_spi.h

C

```
#define DRV_SPI_INDEX_3 3
```

Description

This is macro DRV_SPI_INDEX_3.

DRV_SPI_INDEX_4 Macro

File

drv_spi.h

C

```
#define DRV_SPI_INDEX_4 4
```

Description

This is macro DRV_SPI_INDEX_4.

DRV_SPI_INDEX_5 Macro

File

drv_spi.h

C

```
#define DRV_SPI_INDEX_5 5
```

Description

This is macro DRV_SPI_INDEX_5.

DRV_SPI_TASK_MODE Enumeration

Identifies the various modes of how the tasks function will be run.

File

drv_spi.h

C

```
typedef enum {
  DRV_SPI_TASK_MODE_POLLED,
  DRV_SPI_TASK_MODE_ISR
} DRV_SPI_TASK_MODE;
```

Members

Members	Description
DRV_SPI_TASK_MODE_POLLED	Task is configured to run in polled mode
DRV_SPI_TASK_MODE_ISR	Task is configured to run in interrupt mode

Description

SPI Task Modes Enumeration

This enumeration identifies the various tasks mode

Remarks

None.

DRV_SPI_CLIENT_DATA Structure

Defines the data that can be changed per client.

File

drv_spi.h

C

```
typedef struct _DRV_SPI_CLIENT_DATA {
  uint32_t baudRate;
  DRV_SPI_BUFFER_EVENT_HANDLER operationStarting;
  DRV_SPI_BUFFER_EVENT_HANDLER operationEnded;
} DRV_SPI_CLIENT_DATA;
```

Members

Members	Description
uint32_t baudRate;	Baud Rate Value
DRV_SPI_BUFFER_EVENT_HANDLER operationStarting;	This callback is fired when an operation is about to start on the SPI bus. This allows the user to set any pins that need to be set. This callback may be called from an ISR so should not include OSAL calls. The context parameter is the same one passed into the BufferAddRead, BufferAddWrite, BufferAddWriteRead function.
DRV_SPI_BUFFER_EVENT_HANDLER operationEnded;	This callback is fired when an operation has just completed on the SPI bus. This allows the user to set any pins that need to be set. This callback may be called from an ISR so should not include OSAL calls. The context parameter is the same one passed into the BufferAddRead, BufferAddWrite, BufferAddWriteRead function.

Description

SPI Driver Client Specific Configuration

This data type defines the data can be configured per client. This data can be per client, and overrides the configuration data contained inside of DRV_SPI_INIT.

Remarks

None.

Files

Files

Name	Description
drv_spi.h	SPI device driver interface file.
drv_spi_config_template.h	SPI Driver configuration definitions template.

Description

This section lists the source and header files used by the SPI Driver Library.

drv_spi.h

SPI device driver interface file.

Enumerations

	Name	Description
	DRV_SPI_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_SPI_BUFFER_TYPE	Identifies the various buffer types of the SPI module.
	DRV_SPI_CLOCK_MODE	Identifies the various clock modes of the SPI module.
	DRV_SPI_MODE	Identifies the various usage modes of the SPI module.
	DRV_SPI_PROTOCOL_TYPE	Identifies the various protocols of the SPI module.
	DRV_SPI_TASK_MODE	Identifies the various modes of how the tasks function will be run.

Functions

	Name	Description
	DRV_SPI_BufferAddRead	Registers a buffer for a read operation. Actual transfer will happen in the Task function. **Implementation:** Dynamic
	DRV_SPI_BufferAddRead2	Registers a buffer for a read operation. Actual transfer will happen in the Task function. **Implementation:** Dynamic
	DRV_SPI_BufferAddWrite	Registers a buffer for a write operation. Actual transfer will happen in the Task function. **Implementation:** Dynamic
	DRV_SPI_BufferAddWrite2	Registers a buffer for a write operation. Actual transfer will happen in the Task function. **Implementation:** Dynamic
	DRV_SPI_BufferAddWriteRead	Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddWriteRead2	Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferStatus	Returns the transmitter and receiver transfer status. **Implementation:** Dynamic
	DRV_SPI_ClientConfigure	Configures a SPI client with specific data. **Implementation:** Dynamic
	DRV_SPI_Close	Closes an opened instance of the SPI driver. **Implementation:** Dynamic

© 2013-2015 Microchip Technology Inc.

	DRV_SPI_Deinitialize	Deinitializes the specified instance of the SPI driver module. **Implementation:** Static/Dynamic
	DRV_SPI_Initialize	Initializes the SPI instance for the specified driver index. **Implementation:** Static/Dynamic
	DRV_SPI_Open	Opens the specified SPI driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SPI_Status	Provides the current status of the SPI driver module. **Implementation:** Dynamic
	DRV_SPI_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic
	DRV_SPIn_ReceiverBufferIsFull	Returns the receive buffer status. 'n' represents the instance of the SPI driver used. **Implementation:** Static
	DRV_SPIn_TransmitterBufferIsFull	Returns the transmit buffer status. 'n' represents the instance of the SPI driver used. **Implementation:** Static

Macros

	Name	Description
	DRV_SPI_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_SPI_INDEX_0	SPI driver index definitions.
	DRV_SPI_INDEX_1	This is macro DRV_SPI_INDEX_1.
	DRV_SPI_INDEX_2	This is macro DRV_SPI_INDEX_2.
	DRV_SPI_INDEX_3	This is macro DRV_SPI_INDEX_3.
	DRV_SPI_INDEX_4	This is macro DRV_SPI_INDEX_4.
	DRV_SPI_INDEX_5	This is macro DRV_SPI_INDEX_5.
	DRV_SPI_INDEX_COUNT	Number of valid SPI driver indices.

Structures

	Name	Description
	_DRV_SPI_CLIENT_DATA	Defines the data that can be changed per client.
	_DRV_SPI_INIT	Defines the data required to initialize or reinitialize the SPI driver
	DRV_SPI_CLIENT_DATA	Defines the data that can be changed per client.
	DRV_SPI_INIT	Defines the data required to initialize or reinitialize the SPI driver

Types

	Name	Description
	DRV_SPI_BUFFER_EVENT_HANDLER	Pointer to a SPI Driver Buffer Event handler function
	DRV_SPI_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the driver.

Description

SPI Driver Interface

The SPI driver provides a simple interface to manage the SPI module. This file defines the interface definitions and prototypes for the SPI driver.

File Name

drv_spi.h

Company

Microchip Technology Inc.

drv_spi_config_template.h

SPI Driver configuration definitions template.

Macros

	Name	Description
	DRV_SPI_16BIT	Controls the compilation of 16 Bit mode
	DRV_SPI_32BIT	Controls the compilation of 32 Bit mode
	DRV_SPI_8BIT	Controls the compilation of 8 Bit mode
	DRV_SPI_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_SPI_DMA	Controls the compilation of DMA support
	DRV_SPI_DMA_DUMMY_BUFFER_SIZE	Controls the size of DMA dummy buffer
	DRV_SPI_DMA_TXFER_SIZE	Controls the size of DMA transfers
	DRV_SPI_EBM	Controls the compilation of Enhanced Buffer Mode mode
	DRV_SPI_ELEMENTS_PER_QUEUE	Controls the number of elements that are allocated.
	DRV_SPI_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver .
	DRV_SPI_ISR	Controls the compilation of ISR mode
	DRV_SPI_MASTER	Controls the compilation of master mode
	DRV_SPI_POLLED	Controls the compilation of Polled mode
	DRV_SPI_RM	Controls the compilation of Standard Buffer mode
	DRV_SPI_SLAVE	Controls the compilation of slave mode

Description

SPI Driver Configuration Definitions for the Template Version

These definitions statically define the driver's mode of operation.

File Name

drv_spi_config_template.h

Company

Microchip Technology Inc.

SPI Flash Driver Libraries

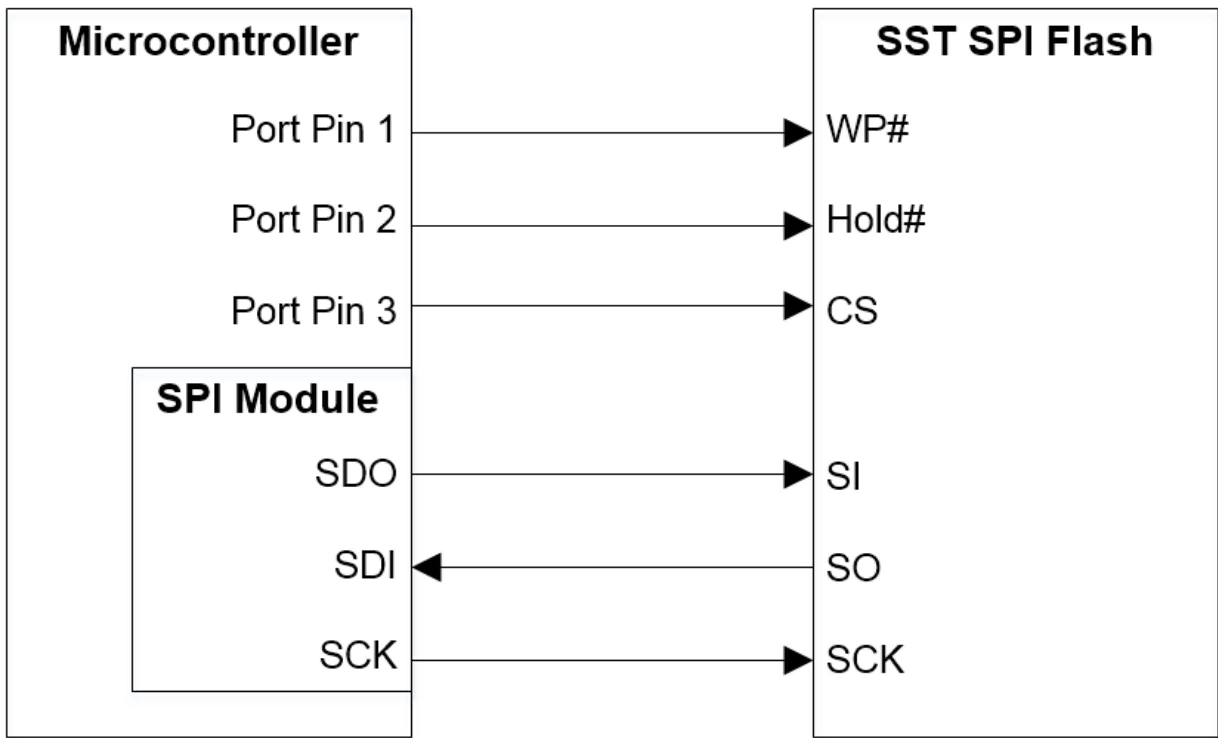
This topic describes the Serial Peripheral Interface (SPI) Flash Driver Libraries.

Introduction

This library provides an interface to manage the SST SPI Flash modules (SST25VF020B, SST25VF016B, and SST25VF064C) in different modes of operation.

Description

The SPI Flash Driver uses SPI interface to establish the communication between SST Flash and Microchip microcontrollers. The SPI module of the controller works as a Master device and the Flash module works as a Slave. The following diagram shows the pin connections that are required to make the driver operational:



The SPI Flash Driver is dynamic in nature, so single instance of it can support multiple clients that want to use the same Flash. Multiple instances of the driver can be used when multiple Flash devices are required to be part of the system. The SPI Driver, which is used by the SPI Flash Driver, can be configured for use in either Polled or Interrupt mode.

Using the Library

This topic describes the basic architecture of the SPI Flash Driver Library and provides information and examples on its use.

Description

Interface Header Files: `drv_sst25vf016b.h`, `drv_sst25vf020b.h`, or `drv_sst25vf064c.h`

The interface to the SPI Flash Driver Library is defined in the header file. Any C language source (`.c`) file that uses the SPI Flash Driver library should include this header.

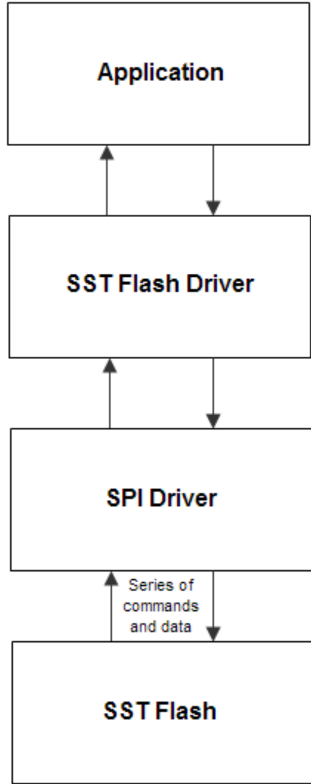
Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the SPI Flash Driver Library with a convenient C language interface. This topic describes how that abstraction is modeled in software.

Description

The SST SPI Flash needs a specific set of commands to be given on its SPI interface along with the required address and data to do different operations. This driver abstracts these requirements and provide simple APIs that can be used to perform Erase, Write, and Read operations. The SPI Driver is used for this purpose. The following layered diagram depicts the communication between different modules.



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall

operation of the SPI Flash module.

Library Interface Section	Description
System Functions	These functions are accessed by the MPLAB Harmony System module and allow the driver to be initialized, deinitialized, and maintained.
Core Client Functions	These functions allow the application client to open and close the driver.
Block Operation Functions	These functions enable the Flash module to be erased, written, and read (to/from).
Media Interface Functions	These functions provide media status and the Flash geometry.

How the Library Works

The library provides interfaces to support:

- System Initialization/Deinitialization
- Opening the Driver
- Block Operations

System Initialization and Deinitialization

Provides information on initializing the system.

Description

System Initialization and Deinitialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization each instance of the SST Flash module would be initialized with the following configuration settings (either passed dynamically at run-time using DRV_SST25VF020B_INIT, DRV_SST25VF016B_INIT, or DRV_SST25VF064C_INIT, or by using Initialization Overrides) that are supported or used by the specific SST Flash device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section
- The SPI Driver Module Index which is intended to be used to communicate with SST Flash (e.g., DRV_SPI_INDEX_0)
- Port Pins of the microcontroller to be used for Chip Select, Write Protection, and Hold operations on the SST Flash device
- Maximum Buffer Queue Size for that instance of the SST Flash Driver

Using the SST25VF020B as an example, the DRV_SST25VF020B_Initialize function returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the Initialize interface would be used by the other system interfaces like DRV_SST25VF020B_Deinitialize, DRV_SST25VF020B_Status, and DRV_SST25VF020B_Tasks.

Note: The system initialization and the deinitialization settings, only affect the instance of the peripheral that is being initialized or deinitialized.

Example:

```
// This code example shows the initialization of the SST25VF020B SPI Flash
// Driver. SPI driver index 0 is used for the purpose. Pin numbers 1, 2,
// and 3 of PORTB are configured for the Hold pin, Write Protection pin, and
// the Chip Select pin, respectively. The maximum buffer queue size is set to 5.

   DRV_SST25VF020B_INIT   SST25VF020BInitData;
   SYS_MODULE_OBJ       objectHandle;

   SST25VF020BInitData.moduleInit.value         = SYS_MODULE_POWER_RUN_FULL;
   SST25VF020BInitData.spiDriverModuleIndex     = DRV_SPI_INDEX_0;
   SST25VF020BInitData.holdPortChannel          = PORT_CHANNEL_B;
   SST25VF020BInitData.holdBitPosition          = PORTS_BIT_POS_1;
   SST25VF020BInitData.writeProtectPortChannel = PORT_CHANNEL_B;
   SST25VF020BInitData.writeProtectBitPosition = PORTS_BIT_POS_2;
   SST25VF020BInitData.chipSelectPortChannel    = PORT_CHANNEL_F;
   SST25VF020BInitData.chipSelectBitPosition    = PORTS_BIT_POS_2;
   SST25VF020BInitData.queueSize                = 5;

   objectHandle = DRV_SST25VF020B_Initialize(DRV_SST25VF020B_INDEX_0,
                                   (SYS_MODULE_INIT*)SST25VF020BInitData);
   if (SYS_MODULE_OBJ_INVALID == objectHandle)
   {
       // Handle error
   }
```

Tasks Routine

The system will either call DRV_SST25VF020B_Tasks, from SYS_Tasks (in a polled environment) or DRV_SST25VF020B_Tasks will be called from the ISR of the SPI module in use.

Opening the Driver

Provides information on opening the driver.

Description

To use the SST Flash driver, the application must open the driver. Using the SST25VF020B as an example, this is done by calling the DRV_SST25VF020B_Open function. Calling this function with DRV_IO_INTENT_NONBLOCKING will cause the driver to be opened in non blocking mode. Then DRV_SST25VF020B_BlockErase, DRV_SST25VF020B_BlockWrite and DRV_SST25VF020B_BlockRead functions when called by this client will be non-blocking.

The client can also open the driver in Read-only mode (DRV_IO_INTENT_READ), Write-only mode (DRV_IO_INTENT_WRITE), and Exclusive mode (DRV_IO_INTENT_EXCLUSIVE). If the driver has been opened exclusively by a client, it cannot be opened again by another client.

If successful, the DRV_SST25VF020B_Open function will return a handle to the driver. This handle records the association between the client and the driver instance that was opened. The DRV_SST25VF020B_Open function may return DRV_HANDLE_INVALID in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return an invalid handle in other (error) cases as well.

The following code shows an example of the driver being opened in different modes.

```
DRV_HANDLE sstHandle1, sstHandle2;

/* Client 1 opens the SST driver in non blocking mode */
sstHandle1 = DRV_SST25VF020B_Open(DRV_SST25VF020B_INDEX_0, DRV_IO_INTENT_NONBLOCKING);

/* Check if the handle is valid */
if(DRV_HANDLE_INVALID == sstHandle1)
{
    /* The driver was not opened successfully. The client
     * can try opening it again */
}

/* Client 2 opens the SST driver in Exclusive Write only mode */
sstHandle2 = DRV_SST25VF020B_Open(DRV_SST25VF020B_INDEX_0, DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);

/* Check if the handle is valid */
if(DRV_HANDLE_INVALID == sstHandle2)
{
    /* The driver was not opened successfully. The client
     * can try opening it again */
}
```

Block Operations

Provides information on block operations.

Description

This driver provides simple client interfaces to Erase, Write, and Read the SST flash in blocks. A block is the unit to represent minimum amount of data that can be erased, written, or read. Block size may differ for Erase, Write, and Read operations. Using the SST25VF020B as an example, the DRV_SST25VF020B_GeometryGet function can be used to determine the different block sizes for the driver.

The DRV_SST25VF020B_BlockErase, DRV_SST25VF020B_BlockWrite, and DRV_SST25VF020B_BlockRead functions are used to erase, write, and read the data to/from SST SPI Flash. These functions are always non-blocking. All of these functions follow a standard queue model to read, write, and erase. When any of these functions are called (i.e., a block request is made), the request is queued. The size of the queue is determined by the queueSize member of the DRV_SST25VF020B_INIT data structure. All of the requests in the queue are executed by the DRV_SST25VF020B_Tasks function one-by-one.

When the driver adds a request to the queue, it returns a buffer handle. This handle allows the client to track the request as it progresses through the queue. The buffer handle expires when the event associated with the buffer completes. The driver provides driver events (DRV_SST25VF020B_BLOCK_EVENT) that indicate termination of the buffer requests.

The following steps can be performed for a simple Block Data Operation:

1. The system should have completed necessary initialization of the SPI Driver and the SST Flash Driver, and the DRV_SST25VF020B_Tasks function should be running in a polled environment.
2. The DRV_SPI_Tasks function should be running in either a polled environment or an interrupt environment.
3. Open the driver using DRV_SST25VF020B_Open with the necessary intent.
4. Set an event handler callback using the function DRV_SST25VF020B_BlockEventHandlerSet.
5. Request for block operations using the functions, DRV_SST25VF020B_BlockErase, DRV_SST25VF020B_BlockWrite, and DRV_SST25VF020B_BlockRead, with the appropriate parameters.
6. Wait for event handler callback to occur and check the status of the block operation using the callback function parameter of type DRV_SST25VF020B_BLOCK_EVENT.
7. The client will be able to close the driver using the function, DRV_SST25VF020B_Close, when required.

Example:

```
/* This code example shows usage of the block operations
 * on the SPI Flash SST25VF020B device */

DRV_HANDLE sstHandle1;
uint8_t myData1[10], myData2[10];
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE  blockHandle1, blockHandle2, blockHandle3;

/* The driver is opened for read-write in Exclusive mode */
sstHandle1 = DRV_SST25VF020B_Open(DRV_SST25VF020B_INDEX_0,
                                  DRV_IO_INTENT_READWRITE | DRV_IO_INTENT_EXCLUSIVE);

/* Check if the driver was opened successfully */
if(DRV_HANDLE_INVALID == sstHandle1)
{
    /* The driver could not be opened successfully */
}

/* Register a Buffer Event Handler with SST25VF020B  driver.
 * This event handler function will be called whenever
 * there is a buffer event.   An application defined
 * context can also be specified. This is returned when
 * the event handler is called.
 * */
DRV_SST25VF020B_BlockEventHandlerSet(sstHandle1,
        APP_SSTBufferEventHandler, NULL);

/* Request for all the three block operations one by one */
```

```
/* first block API to erase 1 block of the flash starting from address 0x0, each block is of 4kbyte */
DRV_SST25VF020B_BlockErase(sstHandle1, &blockHandle1, 0x0, 1);
/* 2nd block API to write myData1 in the first 10 locations of the flash */
DRV_SST25VF020B_BlockWrite(sstHandle1, &blockHandle2, &myData1[0], 0x0, 10);
/* 3rd block API to read the first 10 locations of the flash into myData2 */
DRV_SST25VF020B_BlockRead(sstHandle1, &blockHandle3, &myData2[0], 0x0, 10);

/* This is the Driver Event Handler */

void APP_SSTBufferEventHandler(DRV_SST25VF020B_BLOCK_EVENT event,
        DRV_SST25VF020B_BLOCK_COMMAND_HANDLE blockHandle, uintptr_t contextHandle)
{
    switch(event)
    {
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE:
    if ( blockHandle == blockHandle3)
    {
            /* This means the data was read */
    /* Do data verification/processing */
    }
            break;
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR:
            /* Error handling here. */
            break;
        default:
            break;
    }
}
```

Configuring the Library

SST25VF016B Configuration

	Name	Description
	DRV_SST25VF016B_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_SST25VF016B_HARDWARE_HOLD_ENABLE	Specifies if the hardware hold feature is enabled or not.
	DRV_SST25VF016B_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies if the hardware write protect feature is enabled or not.
	DRV_SST25VF016B_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_SST25VF016B_MODE	Determines whether the driver is implemented as static or dynamic
	DRV_SST25VF016B_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

SST25VF020B Configuration

	Name	Description
	DRV_SST25VF020B_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_SST25VF020B_HARDWARE_HOLD_ENABLE	Specifies if the hardware hold feature is enabled or not.
	DRV_SST25VF020B_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies if the hardware write protect feature is enabled or not.
	DRV_SST25VF020B_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_SST25VF020B_MODE	Determines whether the driver is implemented as static or dynamic.
	DRV_SST25VF020B_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

SST25VF064C Configuration

	Name	Description
	DRV_SST25VF064C_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_SST25VF064C_HARDWARE_HOLD_ENABLE	Specifies whether or not the hardware hold feature is enabled.
	DRV_SST25VF064C_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies if whether or not the hardware write protect feature is enabled.
	DRV_SST25VF064C_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_SST25VF064C_MODE	Determines whether the driver is implemented as static or dynamic.

	DRV_SST25VF064C_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

Description

The SST Flash Driver requires the specification of compile-time configuration macros. These macros define resource usage, feature availability, and dynamic behavior of the driver. These configuration macros should be defined in the `system_config.h` file.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

SST25VF016B Configuration

DRV_SST25VF016B_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_CLIENTS_NUMBER 4
```

Description

SST25VF016B Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. This value represents the total number of clients to be supported across all hardware instances. So if SST25VF016B-1 will be accessed by 2 clients and SST25VF016B-2 will accessed by 3 clients, then this number should be 5. It is recommended that this be set exactly equal to the number of expected clients. Client support consumes RAM memory space. If this macro is not defined and the DRV_SST25VF016B_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - single client operation. If this macro is defined and the DRV_SST25VF016B_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - multi client operation.

Remarks

None.

DRV_SST25VF016B_HARDWARE_HOLD_ENABLE Macro

Specifies if the hardware hold feature is enabled or not.

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_HARDWARE_HOLD_ENABLE false
```

Description

SST25VF016B Hardware HOLD Support

This macro defines if the hardware hold feature is enabled or not. If hardware hold is enabled, then user must provide a port pin corresponding to HOLD pin on the flash

Remarks

None

DRV_SST25VF016B_HARDWARE_WRITE_PROTECTION_ENABLE Macro

Specifies if the hardware write protect feature is enabled or not.

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_HARDWARE_WRITE_PROTECTION_ENABLE false
```

Description

SST25VF016B Hardware Write Protect Support

This macro defines if the hardware Write Protect feature is enabled or not. If hardware write protection is enabled, then user must provide a port pin corresponding to WP pin on the flash

Remarks

None.

DRV_SST25VF016B_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_INSTANCES_NUMBER 2
```

Description

SST25VF016B driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of SST25VF016B modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_SST25VF016B_MODE Macro

Determines whether the driver is implemented as static or dynamic

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_MODE DYNAMIC
```

Description

SST25VF016B mode

Determines whether the driver is implemented as static or dynamic. Static drivers control the peripheral directly with peripheral library routines.

Remarks

None.

© 2013-2015 Microchip Technology Inc.

DRV_SST25VF016B_QUEUE_DEPTH_COMBINED Macro

Number of entries of queues in all instances of the driver.

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_QUEUE_DEPTH_COMBINED 7
```

Description

SST25VF016B Driver Instance combined queue depth.

This macro defines the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for all the read/write/erase operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build).

A buffer queue will contain buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all SST25VF016B driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking erase/write/read requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its buffer queue size.

SST25VF020B Configuration

DRV_SST25VF020B_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_CLIENTS_NUMBER 4
```

Description

SST25VF020B Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. This value represents the total number of clients to be supported across all hardware instances. So if SST25VF020B-1 will be accessed by 2 clients and SST25VF020B-2 will accessed by 3 clients, then this number should be 5. It is recommended that this be set exactly equal to the number of expected clients. Client support consumes RAM memory space. If this macro is not defined and the DRV_SST25VF020B_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - single client operation. If this macro is defined and the DRV_SST25VF020B_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - multi client operation.

Remarks

None.

DRV_SST25VF020B_HARDWARE_HOLD_ENABLE Macro

Specifies if the hardware hold feature is enabled or not.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_HARDWARE_HOLD_ENABLE false
```

Description

SST25VF020B Hardware HOLD Support

This macro defines if the hardware hold feature is enabled or not. If hardware hold is enabled, then user must provide a port pin corresponding to HOLD pin on the flash

Remarks

None.

DRV_SST25VF020B_HARDWARE_WRITE_PROTECTION_ENABLE Macro

Specifies if the hardware write protect feature is enabled or not.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_HARDWARE_WRITE_PROTECTION_ENABLE false
```

Description

SST25VF020B Hardware Write Protect Support

This macro defines if the hardware Write Protect feature is enabled or not. If hardware write protection is enabled, then user must provide a port pin corresponding to WP pin on the flash

Remarks

None.

DRV_SST25VF020B_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_INSTANCES_NUMBER 2
```

Description

SST25VF020B driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of SST25VF020B modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_SST25VF020B_MODE Macro

Determines whether the driver is implemented as static or dynamic.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_MODE DYNAMIC
```

Description

SST25VF020B mode

Determines whether the driver is implemented as static or dynamic. Static drivers control the peripheral directly with peripheral library routines.

Remarks

None.

DRV_SST25VF020B_QUEUE_DEPTH_COMBINED Macro

Number of entries of queues in all instances of the driver.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_QUEUE_DEPTH_COMBINED 7
```

Description

SST25VF020B Driver Instance combined queue depth.

This macro defines the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for all the read/write/erase operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build).

A buffer queue will contain buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all SST25VF020B driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking erase/write/read requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its buffer queue size.

SST25VF064C Configuration

DRV_SST25VF064C_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_CLIENTS_NUMBER 4
```

Description

SST25VF064C Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. This value represents the total number of clients to be supported across all hardware instances. So if SST25VF064C-1 will be accessed by two clients and SST25VF064C-2 will accessed by three clients, this number should be 5. It is recommended that this be set exactly equal to the number of expected clients. Client support consumes RAM memory space. If this macro is not defined and the DRV_SST25VF064C_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - single client operation. If this macro is defined and the DRV_SST25VF064C_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - multi-client operation.

Remarks

None.

DRV_SST25VF064C_HARDWARE_HOLD_ENABLE Macro

Specifies whether or not the hardware hold feature is enabled.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_HARDWARE_HOLD_ENABLE false
```

Description

SST25VF064C Hardware HOLD Support

This macro defines whether or not the hardware hold feature is enabled. If hardware hold is enabled, the user must provide a port pin corresponding to the HOLD pin on the Flash device.

Remarks

None.

DRV_SST25VF064C_HARDWARE_WRITE_PROTECTION_ENABLE Macro

Specifies if whether or not the hardware write protect feature is enabled.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_HARDWARE_WRITE_PROTECTION_ENABLE false
```

Description

SST25VF064C Hardware Write Protect Support

This macro defines whether or not the hardware Write Protect feature is enabled. If hardware write protection is enabled, the user must provide a port pin corresponding to the WP pin on the Flash device.

Remarks

None.

DRV_SST25VF064C_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_INSTANCES_NUMBER 2
```

Description

SST25VF064C driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of SST25VF064C modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, the driver will be built statically.

Remarks

None.

DRV_SST25VF064C_MODE Macro

Determines whether the driver is implemented as static or dynamic.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_MODE DYNAMIC
```

Description

SST25VF064C mode

Determines whether the driver is implemented as static or dynamic. Static drivers control the peripheral directly with peripheral library routines.

Remarks

None.

DRV_SST25VF064C_QUEUE_DEPTH_COMBINED Macro

Number of entries of queues in all instances of the driver.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_QUEUE_DEPTH_COMBINED 7
```

Description

SST25VF064C Driver Instance combined queue depth.

This macro defines the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for all the read/write/erase operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build).

A buffer queue will contain buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all SST25VF064C driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking erase/write/read requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its buffer queue size.

Building the Library

This section lists the files that are available in the SPI Flash Driver Library.

Description

This section list the files that are available in the `/src` folder of the SPI Flash Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/spi_flash`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`sst25vf016b/drv_sst25vf016b.h`	Header file that exports the SST25VF016B driver API.
`sst25vf020b/drv_sst25vf020b.h`	Header file that exports the SST25VF020B driver API.
`sst25vf064c/drv_sst25vf064c.h`	Header file that exports the SST25VF064C driver API.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`sst25vf016b/src/dynamic/drv_sst25vf016b.c`	Basic SPI Flash Driver SST25VF016B implementation file.
`sst25vf020b/src/dynamic/drv_sst25vf020b.c`	Basic SPI Flash Driver SST25VF020B implementation file.
`sst25vf064c/src/dynamic/drv_sst25vf064c.c`	Basic SPI Flash Driver SST25VF064C implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
`sst25vf020b/src/dynamic/drv_sst25vf020b_erasewrite.c`	This file implements an optional BlockEraseWrite feature for the SST25VF020B driver.

Module Dependencies

The SPI Flash Driver Library depends on the following modules:

- SPI Driver Library
- Ports System Service Library

Library Interface

This section describes the API functions of the SPI Flash Driver Library.

Refer to each section for a detailed description.

SST25FV016B API

a) System Functions

	Name	Description
	DRV_SST25VF016B_Initialize	Initializes the SST25VF016B SPI Flash Driver instance for the specified driver index. **Implementation:** Dynamic
	DRV_SST25VF016B_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module. **Implementation:** Dynamic
	DRV_SST25VF016B_Status	Gets the current status of the SPI Flash Driver module. **Implementation:** Dynamic
	DRV_SST25VF016B_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR. **Implementation:** Dynamic

b) Core Client Functions

	Name	Description
	DRV_SST25VF016B_Close	Closes an opened-instance of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF016B_Open	Opens the specified SPI Flash driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SST25VF016B_ClientStatus	Gets current client-specific status of the SPI Flash driver. **Implementation:** Dynamic

c) Block Operation Functions

	Name	Description
	DRV_SST25VF016B_BlockErase	Erase the specified number of blocks in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockRead	Reads blocks of data starting from the specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockWrite	Write blocks of data starting from a specified address in Flash memory. **Implementation:** Dynamic

d) Media Interface Functions

	Name	Description
	DRV_SST25VF016B_GeometryGet	Returns the geometry of the device. **Implementation:** Dynamic

	DRV_SST25VF016B_MediaIsAttached	Returns the status of the media. **Implementation:** Dynamic

e) Data Types and Constants

	Name	Description
	DRV_SST25VF016B_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_SST25VF016B_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF016B_CLIENT_STATUS	Defines the client status. **Implementation:** Dynamic
	DRV_SST25VF016B_EVENT_HANDLER	Pointer to a SST25VF016B SPI Flash Driver Event handler function. **Implementation:** Dynamic
	DRV_SST25VF016B_INIT	Contains all the data necessary to initialize the SPI Flash device. **Implementation:** Dynamic
	DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF016B_INDEX_0	SPI Flash driver index definitions
	DRV_SST25VF016B_INDEX_1	This is macro DRV_SST25VF016B_INDEX_1.

Description

This section contains the SST25V016B Flash device API.

a) System Functions

DRV_SST25VF016B_Initialize Function

Initializes the SST25VF016B SPI Flash Driver instance for the specified driver index.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
SYS_MODULE_OBJ DRV_SST25VF016B_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT *
const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the SPI Flash driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This function must be called before any other SPI Flash function is called.

This function should only be called once during system initialization unless DRV_SST25VF016B_Deinitialize is called to deinitialize the driver instance.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the SST25VF016B SPI
// Flash Driver. SPI driver index 0 is used for the purpose. Pin numbers 1, 2
// and 3 of port channel B are configured for hold pin, write protection pin
// and chip select pin respectively. Maximum buffer queue size is set 5.

DRV_SST25VF016B_INIT   SST25VF016BInitData;
SYS_MODULE_OBJ      objectHandle;

SST25VF016BInitData.moduleInit.value      = SYS_MODULE_POWER_RUN_FULL;
SST25VF016BInitData.spiDriverModuleIndex  = DRV_SPI_INDEX_0;
SST25VF016BInitData.holdPortChannel       = PORT_CHANNEL_B;
SST25VF016BInitData.holdBitPosition       = PORTS_BIT_POS_1;
SST25VF016BInitData.writeProtectPortChannel = PORT_CHANNEL_B;
SST25VF016BInitData.writeProtectBitPosition = PORTS_BIT_POS_2;
SST25VF016BInitData.chipSelectPortChannel = PORT_CHANNEL_F;
SST25VF016BInitData.chipSelectBitPosition = PORTS_BIT_POS_2;
SST25VF016BInitData.queueSize = 5;

objectHandle = DRV_SST25VF016B_Initialize(DRV_SST25VF016B_INDEX_0,
                                (SYS_MODULE_INIT*)SST25VF016BInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_SST25VF016B_Initialize
(
const SYS_MODULE_INDEX index,
const SYS_MODULE_INIT * const init
);

DRV_SST25VF016B_Deinitialize Function

Deinitializes the specified instance of the SPI Flash driver module.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SPI Flash Driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF016B_Initialize should have been called before calling this function.

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_SST25VF016B_Initialize
SYS_STATUS          status;


DRV_SST25VF016B_Deinitialize(object);

status = DRV_SST25VF016B_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF016B_Initialize

Function

void DRV_SST25VF016B_Deinitialize(SYS_MODULE_OBJ object)

DRV_SST25VF016B_Status Function

Gets the current status of the SPI Flash Driver module.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
SYS_STATUS DRV_SST25VF016B_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations

SYS_STATUS_UNINITIALIZED - Indicates that the driver is not initialized

Description

This function provides the current status of the SPI Flash Driver module.

Remarks

A driver can only be opened when its status is SYS_STATUS_READY.

Preconditions

Function DRV_SST25VF016B_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF016B_Initialize
SYS_STATUS          SST25VF016BStatus;

SST25VF016BStatus = DRV_SST25VF016B_Status(object);
else if (SYS_STATUS_ERROR >= SST25VF016BStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF016B_Initialize

Function

SYS_STATUS DRV_SST25VF016B_Status(SYS_MODULE_OBJ object)

DRV_SST25VF016B_Tasks Function

Maintains the driver's read, erase, and write state machine and implements its ISR.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal state machine and should be called from the system's Tasks function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks function (SYS_Tasks).

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI Flash driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF016B_Initialize

while (true)
{
    DRV_SST25VF016B_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SST25VF016B_Initialize)

Function

void DRV_SST25VF016B_Tasks (SYS_MODULE_OBJ object);

b) Core Client Functions

DRV_SST25VF016B_Close Function

Closes an opened-instance of the SPI Flash driver.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened-instance of the SPI Flash driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SST25VF016B_Open before the caller may use the driver again.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SST25VF016B_Open

DRV_SST25VF016B_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_SST25VF016B_Close(DRV_Handle handle);

DRV_SST25VF016B_Open Function

Opens the specified SPI Flash driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
DRV_HANDLE DRV_SST25VF016B_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_SST25VF016B_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver status is not ready.

The driver status becomes ready inside "DRV_SST25VF016B_Tasks" function. To make the SST Driver status ready and hence successfully "Open" the driver, "Task" routine need to be called periodically.

Description

This function opens the specified SPI Flash driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The driver will always work in Non-Blocking mode even if IO-intent is selected as blocking.

The handle returned is valid until the DRV_SST25VF016B_Close function is called.

This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF016B_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_SST25VF016B_Open(DRV_SST25VF016B_INDEX_0,
                                                DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SST25VF016B_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

);

DRV_SST25VF016B_ClientStatus Function

Gets current client-specific status of the SPI Flash driver.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
DRV_SST25VF016B_CLIENT_STATUS DRV_SST25VF016B_ClientStatus(const DRV_HANDLE handle);
```

Returns

A DRV_SST25VF016B_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the SPI Flash driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;         // Returned from DRV_SST25VF016B_Open
DRV_SST25VF016B_CLIENT_STATUS     clientStatus;

clientStatus = DRV_SST25VF016B_ClientStatus(handle);
if(DRV_SST25VF016B_CLIENT_STATUS_READY == clientStatus)
{
    // do the tasks
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's open

Function

DRV_SST25VF016B_CLIENT_STATUS DRV_SST25VF016B_ClientStatus(DRV_HANDLE handle);

c) Block Operation Functions

DRV_SST25VF016B_BlockErase Function

Erase the specified number of blocks in Flash memory.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_BlockErase(const DRV_HANDLE handle, DRV_SST25VF016B_BLOCK_COMMAND_HANDLE *
commandHandle, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It Will be DRV_BUFFER_HANDLE_INVALID if the request was not queued.

Description

This function schedules a non-blocking erase operation in flash memory. The function returns with a valid erase handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the client opened the driver for read only
- if nBlock is 0
- if the queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF016B_EVENT_ERASE_COMPLETE event if the erase operation was successful or DRV_SST25VF016B_EVENT_ERASE_ERROR event if the erase operation was not successful.

Remarks

Write Protection will be disabled for the complete flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF016B_Open call.

Example

```
// Destination address should be block aligned.
uint32_t blockStart;
uint32_t nBlock;
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF016BHandle is the handle returned
// by the DRV_SST25VF016B_Open function.

// Client registers an event handler with driver

DRV_SST25VF016B_BlockEventHandlerSet(mySST25VF016BHandle,
                APP_SST25VF016BEventHandler, (uintptr_t)&myAppObj);
```

```
DRV_SST25VF016B_BlockErase( mySST25VF016BHandle, commandHandle,
                                                    blockStart, nBlock );

if(DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer queue is processed.

void APP_SST25VF016BEventHandler(DRV_SST25VF016B_BLOCK_EVENT event,
        DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF016B_EVENT_ERASE_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF016B_EVENT_ERASE_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
blockStart	Start block address in SST25VF016B memory from where the erase should begin. LSBs (A0-A11) of block start address will be ignored to align it with Erase block size boundary.
nBlock	Total number of blocks to be erased. Each Erase block is of size 4 KByte.

Function

void DRV_SST25VF016B_BlockErase

(

const DRV_HANDLE handle,

DRV_SST25VF016B_BLOCK_COMMAND_HANDLE * commandHandle,

uint32_t blockStart,

uint32_t nBlock

);

DRV_SST25VF016B_BlockEventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_BlockEventHandlerSet(const DRV_HANDLE handle, const
DRV_SST25VF016B_EVENT_HANDLER eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client calls any read, write or erase function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any read/write/erase operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI FLash driver instance.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle;

// mySST25VF016BHandle is the handle returned
// by the DRV_SST25VF016B_Open function.

// Client registers an event handler with driver. This is done once.

DRV_SST25VF016B_BlockEventHandlerSet( mySST25VF016BHandle,
                APP_SST25VF016BEventHandler, (uintptr_t)&myAppObj );

DRV_SST25VF016B_BlockRead( mySST25VF016BHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_SST25VF016BEventHandler(DRV_SST25VF016B_BLOCK_EVENT event,
        DRV_SST25VF016B_BLOCK_COMMAND_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
```

```
    {
        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_SST25VF016B_BlockEventHandlerSet

(

const DRV_HANDLE handle,

const DRV_SST25VF016B_EVENT_HANDLER eventHandler,

const uintptr_t context

);

DRV_SST25VF016B_BlockRead Function

Reads blocks of data starting from the specified address in Flash memory.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_BlockRead(const DRV_HANDLE handle, DRV_SST25VF016B_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * targetBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation for reading blocks of data from flash memory. The function returns with a valid handle in the commandHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the

following circumstances:

- if a buffer could not be allocated to the request
- if the target buffer pointer is NULL
- if the client opened the driver for write only
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully of DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

The maximum read speed is 33 MHz.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF016B_Open call.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF016B_BASE_ADDRESS_TO_READ_FROM;
uint32_t    nBlock = 2;
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF016BHandle is the handle returned
// by the DRV_SST25VF016B_Open function.

// Client registers an event handler with driver

DRV_SST25VF016B_BlockEventHandlerSet(mySST25VF016BHandle,
                APP_SST25VF016BEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF016B_BlockRead( mySST25VF016BHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_SST25VF016BEventHandler(DRV_SST25VF016B_BLOCK_EVENT event,
        DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.
```

```
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
*targetBuffer	Buffer into which the data read from the SPI Flash instance will be placed
blockStart	Start block address in SST25VF016B memory from where the read should begin. It can be any address of the flash.
nBlock	Total number of blocks to be read. Each Read block is of 1 byte.

Function

void DRV_SST25VF016B_BlockRead

(

const DRV_HANDLE handle,

DRV_SST25VF016B_BLOCK_COMMAND_HANDLE * commandHandle,

uint8_t *targetBuffer,

uint32_t blockStart,

uint32_t nBlock

);

DRV_SST25VF016B_BlockWrite Function

Write blocks of data starting from a specified address in Flash memory.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_BlockWrite(DRV_HANDLE handle, DRV_SST25VF016B_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into flash memory. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL

- if the client opened the driver for read only
- if the buffer size is 0
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

In the case of multi bytes write operation, byte by byte writing will happen instead of Address auto Increment writing.

Write Protection will be disabled for the complete flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF016B_Open call.

The flash address location which has to be written, must be erased before using the API DRV_SST25VF016B_BlockErase().

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF016B_BASE_ADDRESS_TO_WRITE_TO;
uint32_t    nBlock = 2;
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF016BHandle is the handle returned
// by the DRV_SST25VF016B_Open function.

// Client registers an event handler with driver

DRV_SST25VF016B_BlockEventHandlerSet(mySST25VF016BHandle,
                APP_SST25VF016BEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF016B_BlockWrite( mySST25VF016BHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_SST25VF016BEventHandler(DRV_SST25VF016B_BLOCK_EVENT event,
        DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR:
```

```
                // Error handling here.

                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function commandHandle -Pointer to an argument that will contain the return buffer handle
sourceBuffer	The source buffer containing data to be programmed into SPI Flash
blockStart	Start block address of SST25VF016B Flash where the write should begin. It can be any address of the flash.
nBlock	Total number of blocks to be written. Each write block is of 1 byte.

Function

void DRV_SST25VF016B_BlockWrite

(

DRV_HANDLE handle,

DRV_SST25VF016B_BLOCK_COMMAND_HANDLE * commandHandle,

uint8_t *sourceBuffer,

uint32_t blockStart,

uint32_t nBlock

);

d) Media Interface Functions

DRV_SST25VF016B_GeometryGet Function

Returns the geometry of the device.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_SST25VF016B_GeometryGet(DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Structure which holds the media geometry information.

Description

This API gives the following geometrical details of the SST25VF016B Flash:

- Media Property
- Number of Read/Write/Erase regions in the flash device
- Number of Blocks and their size in each region of the device

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
SYS_FS_MEDIA_GEOMETRY * sstFlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

sstFlashGeometry = DRV_SST25VF016B_GeometryGet(sstOpenHandle1);

// read block size should be 1 byte
readBlockSize  = sstFlashGeometry->geometryTable->blockSize;
nReadBlocks = sstFlashGeometry->geometryTable->numBlocks;
nReadRegions = sstFlashGeometry->numReadRegions;

// write block size should be 1 byte
writeBlockSize  = (sstFlashGeometry->geometryTable +1)->blockSize;
// erase block size should be 4k byte
eraseBlockSize  = (sstFlashGeometry->geometryTable +2)->blockSize;

// total flash size should be 256k byte
totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY DRV_SST25VF016B_GeometryGet(DRV_HANDLE handle);

DRV_SST25VF016B_MediaIsAttached Function

Returns the status of the media.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
bool DRV_SST25VF016B_MediaIsAttached(DRV_HANDLE handle);
```

Returns

- True - Media is attached
- False - Media is not attached

Description

This API tells if the media is attached or not.

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
if (DRV_SST25VF016B_MediaIsAttached(handle))
{
// Do Something
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SST25VF016B_MediaIsAttached(DRV_HANDLE handle);

e) Data Types and Constants

DRV_SST25VF016B_BLOCK_COMMAND_HANDLE Type

Handle identifying block commands of the driver.

File

drv_sst25vf016b.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_SST25VF016B_BLOCK_COMMAND_HANDLE;
```

Description

SPI Flash Driver Block Command Handle

A block command handle is returned by a call to the Read, Write, or Erase functions. This handle allows the application to track the completion of the operation. The handle is returned back to the client by the "event handler callback" function registered with the driver.

The handle assigned to a client request expires when the client has been notified of the completion of the operation (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SST25VF016B_BLOCK_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_sst25vf016b.h

C

```
typedef enum {
  DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR
} DRV_SST25VF016B_BLOCK_EVENT;
```

Members

Members	Description
DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE	Block operation has been completed successfully. Read/Write/Erase Complete
DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR	There was an error during the block operation Read/Write/Erase Error

Description

SST25VF016B SPI Flash Driver Events

This enumeration identifies the possible events that can result from a Read, Write, or Erase request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SST25VF016B_BlockEventHandlerSet function when a block request is

completed.

DRV_SST25VF016B_CLIENT_STATUS Enumeration

Defines the client status.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
typedef enum {
  DRV_SST25VF016B_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0,
  DRV_SST25VF016B_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  DRV_SST25VF016B_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED,
  DRV_SST25VF016B_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR
} DRV_SST25VF016B_CLIENT_STATUS;
```

Members

Members	Description
DRV_SST25VF016B_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0	Up and running, ready to start new operations
DRV_SST25VF016B_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	Operation in progress, unable to start a new one
DRV_SST25VF016B_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED	Client is closed
DRV_SST25VF016B_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR	Client Error

Description

SPI Flash Client Status

Defines the various client status codes.

Remarks

None.

DRV_SST25VF016B_EVENT_HANDLER Type

Pointer to a SST25VF016B SPI Flash Driver Event handler function.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
typedef void (* DRV_SST25VF016B_EVENT_HANDLER)(DRV_SST25VF016B_BLOCK_EVENT event,
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t context);
```

Returns

None.

Description

SST25VF016B SPI Flash Driver Event Handler Function Pointer

This data type defines the required function signature for the SST25VF016B SPI Flash driver event handling callback

function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values and return value are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE, it means that the data was transferred successfully.

If the event is DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR, it means that the data was not transferred successfully.

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_SST25VF016B_BlockEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The Read, Write, and Erase functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running.

Example

```
void APP_MyBufferEventHandler
(
    DRV_SST25VF016B_BLOCK_EVENT event,
    DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write/Erase requests
context	Value identifying the context of the application that registered the event handling function

DRV_SST25VF016B_INIT Structure

Contains all the data necessary to initialize the SPI Flash device.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX spiDriverModuleIndex;
  PORTS_CHANNEL holdPortChannel;
  PORTS_BIT_POS holdBitPosition;
  PORTS_CHANNEL writeProtectPortChannel;
  PORTS_BIT_POS writeProtectBitPosition;
  PORTS_CHANNEL chipSelectPortChannel;
  PORTS_BIT_POS chipSelectBitPosition;
  uint32_t queueSize;
} DRV_SST25VF016B_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX spiDriverModuleIndex;	Identifies the SPI driver to be used
PORTS_CHANNEL holdPortChannel;	HOLD pin port channel
PORTS_BIT_POS holdBitPosition;	HOLD pin port position
PORTS_CHANNEL writeProtectPortChannel;	Write protect pin port channel
PORTS_BIT_POS writeProtectBitPosition;	Write Protect Bit pin position
PORTS_CHANNEL chipSelectPortChannel;	Chip select pin port channel
PORTS_BIT_POS chipSelectBitPosition;	Chip Select Bit pin position
uint32_t queueSize;	This is the buffer queue size. This is the maximum number of requests that this instance of the driver will queue. For a static build of the driver, this is overridden by the DRV_SST25VF016B_QUEUE_SIZE macro in system_config.h

Description

SST SPI Flash Driver Initialization Data

This structure contains all of the data necessary to initialize the SPI Flash device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_SST25VF016B_Initialize function.

DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID Macro

This value defines the SPI Flash Driver Block Command Invalid handle.

File

drv_sst25vf016b.h

C

```
#define DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID
```

Description

SPI Flash Driver Block Event Invalid Handle

This value defines the SPI Flash Driver Block Command Invalid handle. It is returned by read/write/erase routines when the request could not be taken.

Remarks

None.

DRV_SST25VF016B_INDEX_0 Macro

SPI Flash driver index definitions

File

drv_sst25vf016b.h

C

```
#define DRV_SST25VF016B_INDEX_0 0
```

Description

Driver SPI Flash Module Index reference

These constants provide SST25VF016B SPI Flash driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_SST25VF016B_Initialize and DRV_SST25VF016B_Open routines to identify the driver instance in use.

DRV_SST25VF016B_INDEX_1 Macro

File

drv_sst25vf016b.h

C

```
#define DRV_SST25VF016B_INDEX_1 1
```

Description

This is macro DRV_SST25VF016B_INDEX_1.

SST25VF020B API

a) System Functions

	Name	Description
	DRV_SST25VF020B_Initialize	Initializes the SST25VF020B SPI Flash Driver instance for the specified driver index. **Implementation:** Dynamic
	DRV_SST25VF020B_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module. **Implementation:** Dynamic
	DRV_SST25VF020B_Status	Gets the current status of the SPI Flash Driver module. **Implementation:** Dynamic

	DRV_SST25VF020B_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR. **Implementation:** Dynamic

b) Core Client Functions

	Name	Description
	DRV_SST25VF020B_ClientStatus	Gets current client-specific status of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF020B_CommandStatus	Gets the current status of the command.
	DRV_SST25VF020B_Close	Closes an opened-instance of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF020B_Open	Opens the specified SPI Flash driver instance and returns a handle to it. **Implementation:** Dynamic

c) Block Operation Functions

	Name	Description
	DRV_SST25VF020B_BlockErase	Erase the specified number of blocks in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockRead	Reads blocks of data starting from the specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockWrite	Write blocks of data starting from a specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockEraseWrite	Erase and Write blocks of data starting from a specified address in SST flash memory.

d) Media Interface Functions

	Name	Description
	DRV_SST25VF020B_GeometryGet	Returns the geometry of the device. **Implementation:** Dynamic
	DRV_SST25VF020B_MediaIsAttached	Returns the status of the media. **Implementation:** Dynamic

e) Data Types and Constants

	Name	Description
	DRV_SST25VF020B_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_SST25VF020B_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF020B_CLIENT_STATUS	Defines the client status.
	DRV_SST25VF020B_EVENT_HANDLER	Pointer to a SST25VF020B SPI Flash Driver Event handler function.
	DRV_SST25VF020B_INIT	Contains all the data necessary to initialize the SPI Flash device.
	DRV_SST25VF020B_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.

	DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF020B_INDEX_0	SPI Flash driver index definitions.
	DRV_SST25VF020B_INDEX_1	This is macro DRV_SST25VF020B_INDEX_1.

Description

This section contains the SST25V020B Flash device API.

a) System Functions

DRV_SST25VF020B_Initialize Function

Initializes the SST25VF020B SPI Flash Driver instance for the specified driver index.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
SYS_MODULE_OBJ DRV_SST25VF020B_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT *
const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the SPI Flash driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This function must be called before any other SPI Flash function is called.

This function should only be called once during system initialization unless DRV_SST25VF020B_Deinitialize is called to deinitialize the driver instance.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the SST25VF020B SPI
// Flash Driver. SPI driver index 0 is used for the purpose. Pin numbers 1, 2
// and 3 of port channel B are configured for hold pin, write protection pin
// and chip select pin respectively. Maximum buffer queue size is set 5.

DRV_SST25VF020B_INIT   SST25VF020BInitData;
SYS_MODULE_OBJ      objectHandle;

SST25VF020BInitData.moduleInit.value      = SYS_MODULE_POWER_RUN_FULL;
SST25VF020BInitData.spiDriverModuleIndex  = DRV_SPI_INDEX_0;
SST25VF020BInitData.holdPortChannel       = PORT_CHANNEL_B;
SST25VF020BInitData.holdBitPosition       = PORTS_BIT_POS_1;
SST25VF020BInitData.writeProtectPortChannel = PORT_CHANNEL_B;
SST25VF020BInitData.writeProtectBitPosition = PORTS_BIT_POS_2;
SST25VF020BInitData.chipSelectPortChannel = PORT_CHANNEL_F;
SST25VF020BInitData.chipSelectBitPosition = PORTS_BIT_POS_2;
SST25VF020BInitData.queueSize = 5;

objectHandle = DRV_SST25VF020B_Initialize(DRV_SST25VF020B_INDEX_0,
                                (SYS_MODULE_INIT*)SST25VF020BInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_SST25VF020B_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

);

DRV_SST25VF020B_Deinitialize Function

Deinitializes the specified instance of the SPI Flash driver module.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SPI Flash Driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF020B_Initialize should have been called before calling this function.

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_SST25VF020B_Initialize
SYS_STATUS          status;


DRV_SST25VF020B_Deinitialize(object);

status = DRV_SST25VF020B_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF020B_Initialize

Function

void DRV_SST25VF020B_Deinitialize(SYS_MODULE_OBJ object)

DRV_SST25VF020B_Status Function

Gets the current status of the SPI Flash Driver module.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
SYS_STATUS DRV_SST25VF020B_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations

SYS_STATUS_UNINITIALIZED - Indicates that the driver is not initialized

Description

This function provides the current status of the SPI Flash Driver module.

Remarks

A driver can only be opened when its status is SYS_STATUS_READY.

Preconditions

Function DRV_SST25VF020B_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF020B_Initialize
SYS_STATUS          SST25VF020BStatus;

SST25VF020BStatus = DRV_SST25VF020B_Status(object);
else if (SYS_STATUS_ERROR >= SST25VF020BStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF020B_Initialize

Function

SYS_STATUS DRV_SST25VF020B_Status(SYS_MODULE_OBJ object)

DRV_SST25VF020B_Tasks Function

Maintains the driver's read, erase, and write state machine and implements its ISR.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal state machine and should be called from the system's Tasks function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks function (SYS_Tasks).

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI Flash driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF020B_Initialize

while (true)
{
    DRV_SST25VF020B_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SST25VF020B_Initialize)

Function

void DRV_SST25VF020B_Tasks (SYS_MODULE_OBJ object);

b) Core Client Functions

DRV_SST25VF020B_ClientStatus Function

Gets current client-specific status of the SPI Flash driver.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
DRV_SST25VF020B_CLIENT_STATUS DRV_SST25VF020B_ClientStatus(const DRV_HANDLE handle);
```

Returns

A DRV_SST25VF020B_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the SPI Flash driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;         // Returned from DRV_SST25VF020B_Open
DRV_SST25VF020B_CLIENT_STATUS     clientStatus;

clientStatus = DRV_SST25VF020B_ClientStatus(handle);
if(DRV_SST25VF020B_CLIENT_STATUS_READY == clientStatus)
{
    // do the tasks
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's open

Function

DRV_SST25VF020B_CLIENT_STATUS DRV_SST25VF020B_ClientStatus(DRV_HANDLE handle);

DRV_SST25VF020B_CommandStatus Function

Gets the current status of the command.

File

drv_sst25vf020b.h

C

```
DRV_SST25VF020B_COMMAND_STATUS DRV_SST25VF020B_CommandStatus(const DRV_HANDLE handle, const
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle);
```

Returns

A DRV_SST25VF020B_COMMAND_STATUS value describing the current status of the buffer. Returns DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID if the client handle or the command handle is not valid.

Description

This routine gets the current status of the buffer. The application must use this routine where the status of a scheduled buffer needs to polled on. The function may return DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID in a case where the buffer handle has expired. A buffer handle expires when the internal buffer object is re-assigned to another erase, read or write request. It is recommended that this function be called regularly in order to track the buffer status correctly.

The application can alternatively register an event handler to receive write, read or erase operation completion events.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

Block command request must have been made using Erase, Read or Write APIs to get a valid command handle.

Example

```
DRV_HANDLE      sstOpenHandle;   // Returned from DRV_SST25VF020B_Open
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE  commandHandle;
DRV_SST25VF020B_BlockErase
                        (
                            sstOpenHandle,
                            &commandHandle,
                            0,
                            1
                        );

if(DRV_SST25VF020B_CommandStatus(sstOpenHandle, commandHandle) == DRV_SST25VF020B_COMMAND_COMPLETED );
{
    // do something
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
commandHandle	A valid command handle, returned from Read/Write/Erase APIs.

Function

DRV_SST25VF020B_COMMAND_STATUS DRV_SST25VF020B_CommandStatus

(

const DRV_HANDLE handle,

const DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle

);

DRV_SST25VF020B_Close Function

Closes an opened-instance of the SPI Flash driver.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened-instance of the SPI Flash driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SST25VF020B_Open before the caller may use the driver again.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SST25VF020B_Open

DRV_SST25VF020B_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_SST25VF020B_Close(DRV_Handle handle);

DRV_SST25VF020B_Open Function

Opens the specified SPI Flash driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
DRV_HANDLE DRV_SST25VF020B_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_SST25VF020B_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid

- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver status is not ready.

The driver status becomes ready inside "DRV_SST25VF020B_Tasks" function. To make the SST Driver status ready and hence successfully "Open" the driver, "Task" routine need to be called periodically.

Description

This function opens the specified SPI Flash driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The driver will always work in Non-Blocking mode even if IO-intent is selected as blocking.

The handle returned is valid until the DRV_SST25VF020B_Close function is called.

This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF020B_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_SST25VF020B_Open(DRV_SST25VF020B_INDEX_0,
                                                DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SST25VF020B_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

);

c) Block Operation Functions

DRV_SST25VF020B_BlockErase Function

Erase the specified number of blocks in Flash memory.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_BlockErase(const DRV_HANDLE handle, DRV_SST25VF020B_BLOCK_COMMAND_HANDLE *
commandHandle, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It Will be DRV_BUFFER_HANDLE_INVALID if the request was not queued.

Description

This function schedules a non-blocking erase operation in flash memory. The function returns with a valid erase handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the client opened the driver for read only
- if nBlock is 0
- if the queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF020B_EVENT_ERASE_COMPLETE event if the erase operation was successful or DRV_SST25VF020B_EVENT_ERASE_ERROR event if the erase operation was not successful.

Remarks

Write Protection will be disabled for the complete flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF020B_Open call.

Example

```
// Destination address should be block aligned.
uint32_t blockStart;
uint32_t nBlock;
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF020BHandle is the handle returned
// by the DRV_SST25VF020B_Open function.

// Client registers an event handler with driver

DRV_SST25VF020B_BlockEventHandlerSet(mySST25VF020BHandle,
                APP_SST25VF020BEventHandler, (uintptr_t)&myAppObj);
```

```
DRV_SST25VF020B_BlockErase( mySST25VF020BHandle, commandHandle,
                                                    blockStart, nBlock );

if(DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer queue is processed.

void APP_SST25VF020BEventHandler(DRV_SST25VF020B_BLOCK_EVENT event,
        DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF020B_EVENT_ERASE_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF020B_EVENT_ERASE_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
blockStart	Start block address in SST25VF020B memory from where the erase should begin. LSBs (A0-A11) of block start address will be ignored to align it with Erase block size boundary.
nBlock	Total number of blocks to be erased. Each Erase block is of size 4 KByte.

Function

void DRV_SST25VF020B_BlockErase

(

const DRV_HANDLE handle,

DRV_SST25VF020B_BLOCK_COMMAND_HANDLE * commandHandle,

uint32_t blockStart,

uint32_t nBlock

);

DRV_SST25VF020B_BlockEventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_BlockEventHandlerSet(const DRV_HANDLE handle, const
DRV_SST25VF020B_EVENT_HANDLER eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client calls any read, write or erase function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any read/write/erase operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI FLash driver instance.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle;

// mySST25VF020BHandle is the handle returned
// by the DRV_SST25VF020B_Open function.

// Client registers an event handler with driver. This is done once.

DRV_SST25VF020B_BlockEventHandlerSet( mySST25VF020BHandle,
                APP_SST25VF020BEventHandler, (uintptr_t)&myAppObj );

DRV_SST25VF020B_BlockRead( mySST25VF020BHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_SST25VF020BEventHandler(DRV_SST25VF020B_BLOCK_EVENT event,
        DRV_SST25VF020B_BLOCK_COMMAND_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
```

```
    {
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_SST25VF020B_BlockEventHandlerSet

(

const DRV_HANDLE handle,

const DRV_SST25VF020B_EVENT_HANDLER eventHandler,

const uintptr_t context

);

DRV_SST25VF020B_BlockRead Function

Reads blocks of data starting from the specified address in Flash memory.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_BlockRead(const DRV_HANDLE handle, DRV_SST25VF020B_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * targetBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation for reading blocks of data from flash memory. The function returns with a valid handle in the commandHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the

following circumstances:

- if a buffer could not be allocated to the request
- if the target buffer pointer is NULL
- if the client opened the driver for write only
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully of DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

The maximum read speed is 33 MHz.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF020B_Open call.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF020B_BASE_ADDRESS_TO_READ_FROM;
uint32_t    nBlock = 2;
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF020BHandle is the handle returned
// by the DRV_SST25VF020B_Open function.

// Client registers an event handler with driver

DRV_SST25VF020B_BlockEventHandlerSet(mySST25VF020BHandle,
                APP_SST25VF020BEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF020B_BlockRead( mySST25VF020BHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_SST25VF020BEventHandler(DRV_SST25VF020B_BLOCK_EVENT event,
        DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.
```

```
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
*targetBuffer	Buffer into which the data read from the SPI Flash instance will be placed
blockStart	Start block address in SST25VF020B memory from where the read should begin. It can be any address of the flash.
nBlock	Total number of blocks to be read. Each Read block is of 1 byte.

Function

void DRV_SST25VF020B_BlockRead

(

const DRV_HANDLE handle,

DRV_SST25VF020B_BLOCK_COMMAND_HANDLE * commandHandle,

uint8_t *targetBuffer,

uint32_t blockStart,

uint32_t nBlock

);

DRV_SST25VF020B_BlockWrite Function

Write blocks of data starting from a specified address in Flash memory.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_BlockWrite(DRV_HANDLE handle, DRV_SST25VF020B_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into flash memory. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL

- if the client opened the driver for read only
- if the buffer size is 0
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

In the case of multi bytes write operation, byte by byte writing will happen instead of Address auto Increment writing.

Write Protection will be disabled for the complete flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF020B_Open call.

The flash address location which has to be written, must be erased before using the API DRV_SST25VF020B_BlockErase().

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF020B_BASE_ADDRESS_TO_WRITE_TO;
uint32_t    nBlock = 2;
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF020BHandle is the handle returned
// by the DRV_SST25VF020B_Open function.

// Client registers an event handler with driver

DRV_SST25VF020B_BlockEventHandlerSet(mySST25VF020BHandle,
                APP_SST25VF020BEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF020B_BlockWrite( mySST25VF020BHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_SST25VF020BEventHandler(DRV_SST25VF020B_BLOCK_EVENT event,
        DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR:
```

```
                // Error handling here.

                break;

            default:
                break;
        }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function commandHandle -Pointer to an argument that will contain the return buffer handle
sourceBuffer	The source buffer containing data to be programmed into SPI Flash
blockStart	Start block address of SST25VF020B Flash where the write should begin. It can be any address of the flash.
nBlock	Total number of blocks to be written. Each write block is of 1 byte.

Function

void DRV_SST25VF020B_BlockWrite

(

DRV_HANDLE handle,

DRV_SST25VF020B_BLOCK_COMMAND_HANDLE * commandHandle,

uint8_t *sourceBuffer,

uint32_t blockStart,

uint32_t nBlock

);

DRV_SST25VF020B_BlockEraseWrite Function

Erase and Write blocks of data starting from a specified address in SST flash memory.

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_BlockEraseWrite(DRV_HANDLE hClient, DRV_SST25VF020B_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Description

This function combines the step of erasing blocks of SST Flash and then writing the data. The application can use this function if it wants to avoid having to explicitly delete a block in order to update the bytes contained in the block.

This function schedules a non-blocking operation to erase and write blocks of data into SST flash. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only
- if the buffer size is 0

- if the queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SST25VF020B_EVENT_ERASE_ERROR event if the buffer was not processed successfully.

Remarks

Refer to drv_sst25vf020b.h for usage information.

Function

```
void DRV_SST25VF020B_BlockEraseWrite
(
const       DRV_HANDLE handle,
      DRV_SST25VF020B_BLOCK_COMMAND_HANDLE * commandHandle,
void * sourceBuffer,
uint32_t writeBlockStart,
uint32_t nWriteBlock
)
```

d) Media Interface Functions

DRV_SST25VF020B_GeometryGet Function

Returns the geometry of the device.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_SST25VF020B_GeometryGet(DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Structure which holds the media geometry information.

Description

This API gives the following geometrical details of the SST25VF020B Flash:

- Media Property
- Number of Read/Write/Erase regions in the flash device
- Number of Blocks and their size in each region of the device

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
SYS_FS_MEDIA_GEOMETRY * sstFlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

sstFlashGeometry = DRV_SST25VF020B_GeometryGet(sstOpenHandle1);

// read block size should be 1 byte
readBlockSize  = sstFlashGeometry->geometryTable->blockSize;
nReadBlocks = sstFlashGeometry->geometryTable->numBlocks;
nReadRegions = sstFlashGeometry->numReadRegions;

// write block size should be 1 byte
writeBlockSize  = (sstFlashGeometry->geometryTable +1)->blockSize;
// erase block size should be 4k byte
eraseBlockSize  = (sstFlashGeometry->geometryTable +2)->blockSize;

// total flash size should be 256k byte
totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY DRV_SST25VF020B_GeometryGet(DRV_HANDLE handle);

DRV_SST25VF020B_MediaIsAttached Function

Returns the status of the media.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
bool DRV_SST25VF020B_MediaIsAttached(DRV_HANDLE handle);
```

Returns

- True - Media is attached
- False - Media is not attached

Description

This function determines whether or not the media is attached.

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
if (DRV_SST25VF020B_MediaIsAttached(handle))
{
// Do Something
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SST25VF020B_MediaIsAttached(DRV_HANDLE handle);

e) Data Types and Constants

DRV_SST25VF020B_BLOCK_COMMAND_HANDLE Type

Handle identifying block commands of the driver.

File

drv_sst25vf020b.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_SST25VF020B_BLOCK_COMMAND_HANDLE;
```

Description

SPI Flash Driver Block Command Handle

A block command handle is returned by a call to the Read, Write, or Erase functions. This handle allows the application to track the completion of the operation. The handle is returned back to the client by the "event handler callback" function registered with the driver.

The handle assigned to a client request expires when the client has been notified of the completion of the operation (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SST25VF020B_BLOCK_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_sst25vf020b.h

C

```
typedef enum {
  DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR
} DRV_SST25VF020B_BLOCK_EVENT;
```

Members

Members	Description
DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE	Block operation has been completed successfully. Read/Write/Erase Complete
DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR	There was an error during the block operation Read/Write/Erase Error

Description

SST25VF020B SPI Flash Driver Events

This enumeration identifies the possible events that can result from a Read, Write, or Erase request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SST25VF020B_BlockEventHandlerSet function when a block request is

completed.

DRV_SST25VF020B_CLIENT_STATUS Enumeration

Defines the client status.

File

drv_sst25vf020b.h

C

```
typedef enum {
  DRV_SST25VF020B_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0,
  DRV_SST25VF020B_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  DRV_SST25VF020B_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED,
  DRV_SST25VF020B_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR
} DRV_SST25VF020B_CLIENT_STATUS;
```

Members

Members	Description
DRV_SST25VF020B_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0	Up and running, ready to start new operations
DRV_SST25VF020B_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	Operation in progress, unable to start a new one
DRV_SST25VF020B_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED	Client is closed
DRV_SST25VF020B_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR	Client Error

Description

SPI Flash Client Status

Defines the various client status codes.

Remarks

None.

DRV_SST25VF020B_EVENT_HANDLER Type

Pointer to a SST25VF020B SPI Flash Driver Event handler function.

File

drv_sst25vf020b.h

C

```
typedef void (* DRV_SST25VF020B_EVENT_HANDLER)(DRV_SST25VF020B_BLOCK_EVENT event,
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t context);
```

Returns

None.

Description

SST25VF020B SPI Flash Driver Event Handler Function Pointer

This data type defines the required function signature for the SST25VF020B SPI Flash driver event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the

driver.

The parameters and return values and return value are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE, it means that the data was transferred successfully.

If the event is DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR, it means that the data was not transferred successfully.

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_SST25VF020B_BlockEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The Read, Write, and Erase functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running.

Example

```
void APP_MyBufferEventHandler
(
    DRV_SST25VF020B_BLOCK_EVENT event,
    DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write/Erase requests
context	Value identifying the context of the application that registered the event handling function

DRV_SST25VF020B_INIT Structure

Contains all the data necessary to initialize the SPI Flash device.

File

drv_sst25vf020b.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX spiDriverModuleIndex;
  PORTS_CHANNEL holdPortChannel;
  PORTS_BIT_POS holdBitPosition;
  PORTS_CHANNEL writeProtectPortChannel;
  PORTS_BIT_POS writeProtectBitPosition;
  PORTS_CHANNEL chipSelectPortChannel;
  PORTS_BIT_POS chipSelectBitPosition;
  uint32_t queueSize;
} DRV_SST25VF020B_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX spiDriverModuleIndex;	Identifies the SPI driver to be used
PORTS_CHANNEL holdPortChannel;	HOLD pin port channel
PORTS_BIT_POS holdBitPosition;	HOLD pin port position
PORTS_CHANNEL writeProtectPortChannel;	Write protect pin port channel
PORTS_BIT_POS writeProtectBitPosition;	Write Protect Bit pin position
PORTS_CHANNEL chipSelectPortChannel;	Chip select pin port channel
PORTS_BIT_POS chipSelectBitPosition;	Chip Select Bit pin position
uint32_t queueSize;	This is the buffer queue size. This is the maximum number of requests that this instance of the driver will queue. For a static build of the driver, this is overridden by the DRV_SST25VF020B_QUEUE_SIZE macro in system_config.h

Description

SST SPI Flash Driver Initialization Data

This structure contains all of the data necessary to initialize the SPI Flash device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_SST25VF020B_Initialize function.

DRV_SST25VF020B_COMMAND_STATUS Enumeration

Specifies the status of the command for the read, write and erase operations.

File

drv_sst25vf020b.h

C

```
typedef enum {
  DRV_SST25VF020B_COMMAND_COMPLETED,
  DRV_SST25VF020B_COMMAND_QUEUED,
  DRV_SST25VF020B_COMMAND_IN_PROGRESS,
  DRV_SST25VF020B_COMMAND_ERROR_UNKNOWN
} DRV_SST25VF020B_COMMAND_STATUS;
```

Members

Members	Description
DRV_SST25VF020B_COMMAND_COMPLETED	Requested operation is completed
DRV_SST25VF020B_COMMAND_QUEUED	Scheduled but not started
DRV_SST25VF020B_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_SST25VF020B_COMMAND_ERROR_UNKNOWN	Unknown Command

Description

SST Flash Driver Command Status

SST Flash Driver command Status

This type specifies the status of the command for the read, write and erase operations.

Remarks

None.

DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID Macro

This value defines the SPI Flash Driver Block Command Invalid handle.

File

drv_sst25vf020b.h

C

```
#define DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID
```

Description

SPI Flash Driver Block Event Invalid Handle

This value defines the SPI Flash Driver Block Command Invalid handle. It is returned by read/write/erase routines when the request could not be taken.

Remarks

None.

DRV_SST25VF020B_INDEX_0 Macro

SPI Flash driver index definitions.

File

drv_sst25vf020b.h

C

```
#define DRV_SST25VF020B_INDEX_0 0
```

Description

Driver SPI Flash Module Index reference

These constants provide SST25VF020B SPI Flash driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_SST25VF020B_Initialize and DRV_SST25VF020B_Open routines to

identify the driver instance in use.

DRV_SST25VF020B_INDEX_1 Macro

File

drv_sst25vf020b.h

C

```
#define DRV_SST25VF020B_INDEX_1 1
```

Description

This is macro DRV_SST25VF020B_INDEX_1.

SST25VF064C API

a) System Functions

	Name	Description
	DRV_SST25VF064C_Initialize	Initializes the SST25VF064C SPI Flash Driver instance for the specified driver index.
	DRV_SST25VF064C_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module.
	DRV_SST25VF064C_Status	Gets the current status of the SPI Flash Driver module.
	DRV_SST25VF064C_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR.

b) Core Client Functions

	Name	Description
	DRV_SST25VF064C_ClientStatus	Gets current client-specific status of the SPI Flash driver.
	DRV_SST25VF064C_Close	Closes an opened-instance of the SPI Flash driver.
	DRV_SST25VF064C_CommandStatus	Gets the current status of the command.
	DRV_SST25VF064C_Open	Opens the specified SPI Flash driver instance and returns a handle to it.

c) Block Operation Functions

	Name	Description
	DRV_SST25VF064C_BlockErase	Erase the specified number of blocks in Flash memory.
	DRV_SST25VF064C_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.
	DRV_SST25VF064C_BlockRead	Reads blocks of data starting from the specified address in Flash memory.
	DRV_SST25VF064C_BlockWrite	Write blocks of data starting from a specified address in Flash memory.

d) Media Interface Functions

	Name	Description
	DRV_SST25VF064C_GeometryGet	Returns the geometry of the device.
	DRV_SST25VF064C_MediaIsAttached	Returns the status of the media.

e) Data Types and Constants

	Name	Description
	DRV_SST25VF064C_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_SST25VF064C_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF064C_CLIENT_STATUS	Defines the client status.
	DRV_SST25VF064C_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.
	DRV_SST25VF064C_EVENT_HANDLER	Pointer to a SST25VF064C SPI Flash Driver Event handler function.
	DRV_SST25VF064C_INIT	Contains all the data necessary to initialize the SPI Flash device.
	DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF064C_INDEX_0	SPI Flash driver index definitions.
	DRV_SST25VF064C_INDEX_1	This is macro DRV_SST25VF064C_INDEX_1.

Description

a) System Functions

DRV_SST25VF064C_Initialize Function

Initializes the SST25VF064C SPI Flash Driver instance for the specified driver index.

File

drv_sst25vf064c.h

C

```
SYS_MODULE_OBJ DRV_SST25VF064C_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT *
const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the SPI Flash driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This function must be called before any other SPI Flash function is called. This function should only be called once during system initialization unless DRV_SST25VF064C_Deinitialize is called to deinitialize the driver instance. Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the SST25VF064C SPI
// Flash Driver. SPI driver index 0 is used for the purpose. Pin numbers 1, 2
// and 3 of port channel B are configured for hold pin, write protection pin
// and chip select pin respectively. Maximum buffer queue size is set 5.

DRV_SST25VF064C_INIT   SST25VF064CInitData;
SYS_MODULE_OBJ      objectHandle;

SST25VF064CInitData.moduleInit.value      = SYS_MODULE_POWER_RUN_FULL;
SST25VF064CInitData.spiDriverModuleIndex  = DRV_SPI_INDEX_0;
SST25VF064CInitData.holdPortChannel       = PORT_CHANNEL_B;
SST25VF064CInitData.holdBitPosition       = PORTS_BIT_POS_1;
SST25VF064CInitData.writeProtectPortChannel = PORT_CHANNEL_B;
SST25VF064CInitData.writeProtectBitPosition = PORTS_BIT_POS_2;
SST25VF064CInitData.chipSelectPortChannel = PORT_CHANNEL_F;
SST25VF064CInitData.chipSelectBitPosition = PORTS_BIT_POS_2;
SST25VF064CInitData.queueSize = 5;

objectHandle = DRV_SST25VF064C_Initialize(DRV_SST25VF064C_INDEX_0,
                                (SYS_MODULE_INIT*)SST25VF064CInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized

init	Pointer to a data structure containing data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_SST25VF064C_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

);

DRV_SST25VF064C_Deinitialize Function

Deinitializes the specified instance of the SPI Flash driver module.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SPI Flash Driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF064C_Initialize should have been called before calling this function.

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_SST25VF064C_Initialize
SYS_STATUS          status;


DRV_SST25VF064C_Deinitialize(object);

status = DRV_SST25VF064C_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF064C_Initialize

Function

void DRV_SST25VF064C_Deinitialize(SYS_MODULE_OBJ object)

DRV_SST25VF064C_Status Function

Gets the current status of the SPI Flash Driver module.

File

drv_sst25vf064c.h

C

```
SYS_STATUS DRV_SST25VF064C_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations

SYS_STATUS_UNINITIALIZED - Indicates that the driver is not initialized

Description

This function provides the current status of the SPI Flash Driver module.

Remarks

A driver can only be opened when its status is SYS_STATUS_READY.

Preconditions

Function DRV_SST25VF064C_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF064C_Initialize
SYS_STATUS          SST25VF064CStatus;

SST25VF064CStatus = DRV_SST25VF064C_Status(object);
else if (SYS_STATUS_ERROR >= SST25VF064CStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF064C_Initialize

Function

SYS_STATUS DRV_SST25VF064C_Status(SYS_MODULE_OBJ object)

DRV_SST25VF064C_Tasks Function

Maintains the driver's read, erase, and write state machine and implements its ISR.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal state machine and should be called from the system's Tasks function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks function (SYS_Tasks).

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI Flash driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF064C_Initialize

while (true)
{
    DRV_SST25VF064C_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SST25VF064C_Initialize)

Function

void DRV_SST25VF064C_Tasks (SYS_MODULE_OBJ object);

b) Core Client Functions

DRV_SST25VF064C_ClientStatus Function

Gets current client-specific status of the SPI Flash driver.

File

drv_sst25vf064c.h

C

```
DRV_SST25VF064C_CLIENT_STATUS DRV_SST25VF064C_ClientStatus(const DRV_HANDLE handle);
```

Returns

A DRV_SST25VF064C_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the SPI Flash driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;         // Returned from DRV_SST25VF064C_Open
DRV_SST25VF064C_CLIENT_STATUS     clientStatus;

clientStatus = DRV_SST25VF064C_ClientStatus(handle);
if(DRV_SST25VF064C_CLIENT_STATUS_READY == clientStatus)
{
    // do the tasks
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's open

Function

DRV_SST25VF064C_CLIENT_STATUS DRV_SST25VF064C_ClientStatus(DRV_HANDLE handle);

DRV_SST25VF064C_Close Function

Closes an opened-instance of the SPI Flash driver.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened-instance of the SPI Flash driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SST25VF064C_Open before the caller may use the driver again.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SST25VF064C_Open

DRV_SST25VF064C_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_SST25VF064C_Close(DRV_Handle handle);

DRV_SST25VF064C_CommandStatus Function

Gets the current status of the command.

File

drv_sst25vf064c.h

C

```
DRV_SST25VF064C_COMMAND_STATUS DRV_SST25VF064C_CommandStatus(const DRV_HANDLE handle, const
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle);
```

Returns

A DRV_SST25VF064C_COMMAND_STATUS value describing the current status of the buffer. Returns DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID if the client handle or the command handle is not valid.

Description

This routine gets the current status of the buffer. The application must use this routine where the status of a scheduled buffer needs to polled on. The function may return DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID in a case where the buffer handle has expired. A buffer handle expires when the internal buffer object is re-assigned to another erase, read or write request. It is recommended that this function be called regularly in order to track the buffer status correctly.

The application can alternatively register an event handler to receive write, read or erase operation completion events.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

Block command request must have been made using Erase, Read, or Write APIs to get a valid command handle.

Example

```
DRV_HANDLE       sstOpenHandle;   // Returned from DRV_SST25VF064C_Open
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE  commandHandle;
DRV_SST25VF064C_BlockErase
                        (
                            sstOpenHandle,
                            &commandHandle,
                            0,
                            1
                        );

if(DRV_SST25VF064C_CommandStatus(sstOpenHandle, commandHandle) == DRV_SST25VF064C_COMMAND_COMPLETED );
{
    // do something
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
commandHandle	A valid command handle, returned from Read/Write/Erase APIs.

Function

DRV_SST25VF064C_COMMAND_STATUS DRV_SST25VF064C_CommandStatus

(

const DRV_HANDLE handle,

const DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle

);

DRV_SST25VF064C_Open Function

Opens the specified SPI Flash driver instance and returns a handle to it.

File

drv_sst25vf064c.h

C

```
DRV_HANDLE DRV_SST25VF064C_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_SST25VF064C_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.

Description

This function opens the specified SPI Flash driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The driver will always work in Non-Blocking mode even if IO-intent is selected as blocking.

The handle returned is valid until the DRV_SST25VF064C_Close function is called.

This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF064C_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_SST25VF064C_Open(DRV_SST25VF064C_INDEX_0,
                                                DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SST25VF064C_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

);

c) Block Operation Functions

DRV_SST25VF064C_BlockErase Function

Erase the specified number of blocks in Flash memory.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_BlockErase(const DRV_HANDLE handle, DRV_SST25VF064C_BLOCK_COMMAND_HANDLE *
commandHandle, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It Will be DRV_BUFFER_HANDLE_INVALID if the request was not queued.

Description

This function schedules a non-blocking erase operation in Flash memory. The function returns with a valid erase handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the client opened the driver for read only
- if nBlock is 0
- if the queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF064C_EVENT_ERASE_COMPLETE event if the erase operation was successful or DRV_SST25VF064C_EVENT_ERASE_ERROR event if the erase operation was not successful.

Remarks

Write Protection will be disabled for the complete Flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF064C_Open call.

Example

```
// Destination address should be block aligned.
uint32_t blockStart;
uint32_t nBlock;
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF064CHandle is the handle returned
// by the DRV_SST25VF064C_Open function.

// Client registers an event handler with driver

DRV_SST25VF064C_BlockEventHandlerSet(mySST25VF064CHandle,
                APP_SST25VF064CEventHandler, (uintptr_t)&myAppObj);
```

```
DRV_SST25VF064C_BlockErase( mySST25VF064CHandle, commandHandle,
                                                    blockStart, nBlock );

if(DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer queue is processed.

void APP_SST25VF064CEventHandler(DRV_SST25VF064C_BLOCK_EVENT event,
        DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF064C_EVENT_ERASE_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF064C_EVENT_ERASE_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
blockStart	Start block address in SST25VF064C memory from where the erase should begin. LSBs (A0-A11) of block start address will be ignored to align it with Erase block size boundary.
nBlock	Total number of blocks to be erased. Each Erase block is of size 4 KByte.

Function

void DRV_SST25VF064C_BlockErase

(

const DRV_HANDLE handle,

DRV_SST25VF064C_BLOCK_COMMAND_HANDLE * commandHandle,

uint32_t blockStart,

uint32_t nBlock

);

DRV_SST25VF064C_BlockEventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_BlockEventHandlerSet(const DRV_HANDLE handle, const
DRV_SST25VF064C_EVENT_HANDLER eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client calls any read, write or erase function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any read/write/erase operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI FLash driver instance.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle;

// mySST25VF064CHandle is the handle returned
// by the DRV_SST25VF064C_Open function.

// Client registers an event handler with driver. This is done once.

DRV_SST25VF064C_BlockEventHandlerSet( mySST25VF064CHandle,
                APP_SST25VF064CEventHandler, (uintptr_t)&myAppObj );

DRV_SST25VF064C_BlockRead( mySST25VF064CHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_SST25VF064CEventHandler(DRV_SST25VF064C_BLOCK_EVENT event,
        DRV_SST25VF064C_BLOCK_COMMAND_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
```

```
            break;

        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_SST25VF064C_BlockEventHandlerSet

(

const DRV_HANDLE handle,

const DRV_SST25VF064C_EVENT_HANDLER eventHandler,

const uintptr_t context

);

DRV_SST25VF064C_BlockRead Function

Reads blocks of data starting from the specified address in Flash memory.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_BlockRead(const DRV_HANDLE handle, DRV_SST25VF064C_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * targetBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation for reading blocks of data from Flash memory. The function returns with a valid handle in the commandHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the target buffer pointer is NULL
- if the client opened the driver for write only

- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully of DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

The maximum read speed is 33 MHz.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF064C_Open call.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF064C_BASE_ADDRESS_TO_READ_FROM;
uint32_t    nBlock = 2;
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF064CHandle is the handle returned
// by the DRV_SST25VF064C_Open function.

// Client registers an event handler with driver

DRV_SST25VF064C_BlockEventHandlerSet(mySST25VF064CHandle,
                APP_SST25VF064CEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF064C_BlockRead( mySST25VF064CHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_SST25VF064CEventHandler(DRV_SST25VF064C_BLOCK_EVENT event,
        DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
```

}

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
*targetBuffer	Buffer into which the data read from the SPI Flash instance will be placed
blockStart	Start block address in SST25VF064C memory from where the read should begin. It can be any address of the Flash.
nBlock	Total number of blocks to be read. Each Read block is of 1 byte.

Function

void DRV_SST25VF064C_BlockRead

(

const DRV_HANDLE handle,

DRV_SST25VF064C_BLOCK_COMMAND_HANDLE * commandHandle,

uint8_t *targetBuffer,

uint32_t blockStart,

uint32_t nBlock

);

DRV_SST25VF064C_BlockWrite Function

Write blocks of data starting from a specified address in Flash memory.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_BlockWrite(DRV_HANDLE handle, DRV_SST25VF064C_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into Flash memory. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only
- if the buffer size is 0
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully or

DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

In the case of multi bytes write operation, byte by byte writing will happen instead of Address auto Increment writing.

Write Protection will be disabled for the complete Flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF064C_Open call.

The Flash address location which has to be written, must be erased before using the API DRV_SST25VF064C_BlockErase().

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF064C_BASE_ADDRESS_TO_WRITE_TO;
uint32_t    nBlock = 2;
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF064CHandle is the handle returned
// by the DRV_SST25VF064C_Open function.

// Client registers an event handler with driver

DRV_SST25VF064C_BlockEventHandlerSet(mySST25VF064CHandle,
                APP_SST25VF064CEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF064C_BlockWrite( mySST25VF064CHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_SST25VF064CEventHandler(DRV_SST25VF064C_BLOCK_EVENT event,
        DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function commandHandle -Pointer to an argument that will contain the return buffer handle
sourceBuffer	The source buffer containing data to be programmed into SPI Flash
blockStart	Start block address of SST25VF064C Flash where the write should begin. It can be any address of the Flash.
nBlock	Total number of blocks to be written. Each write block is of 1 byte.

Function

```
void DRV_SST25VF064C_BlockWrite
(
    DRV_HANDLE handle,
    DRV_SST25VF064C_BLOCK_COMMAND_HANDLE * commandHandle,
uint8_t *sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
);
```

d) Media Interface Functions

DRV_SST25VF064C_GeometryGet Function

Returns the geometry of the device.

File

drv_sst25vf064c.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_SST25VF064C_GeometryGet(DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Structure which holds the media geometry information.

Description

This API gives the following geometrical details of the SST25VF064C Flash:

- Media Property
- Number of Read/Write/Erase regions in the Flash device
- Number of Blocks and their size in each region of the device

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
SYS_FS_MEDIA_GEOMETRY * sstFlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

sstFlashGeometry = DRV_SST25VF064C_GeometryGet(sstOpenHandle1);

// read block size should be 1 byte
readBlockSize  = sstFlashGeometry->geometryTable->blockSize;
nReadBlocks = sstFlashGeometry->geometryTable->numBlocks;
nReadRegions = sstFlashGeometry->numReadRegions;

// write block size should be 1 byte
writeBlockSize  = (sstFlashGeometry->geometryTable +1)->blockSize;
// erase block size should be 4k byte
eraseBlockSize  = (sstFlashGeometry->geometryTable +2)->blockSize;

// total Flash size should be 8 MB
totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY DRV_SST25VF064C_GeometryGet(DRV_HANDLE handle);

DRV_SST25VF064C_MediaIsAttached Function

Returns the status of the media.

File

drv_sst25vf064c.h

C

```
bool DRV_SST25VF064C_MediaIsAttached(DRV_HANDLE handle);
```

Returns

- True - Media is attached
- False - Media is not attached

Description

This function determines whether or not the media is attached.

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
if (DRV_SST25VF064C_MediaIsAttached(handle))
{
// Do Something
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SST25VF064C_MediaIsAttached(DRV_HANDLE handle);

e) Data Types and Constants

DRV_SST25VF064C_BLOCK_COMMAND_HANDLE Type

Handle identifying block commands of the driver.

File

drv_sst25vf064c.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_SST25VF064C_BLOCK_COMMAND_HANDLE;
```

Description

SPI Flash Driver Block Command Handle

A block command handle is returned by a call to the Read, Write, or Erase functions. This handle allows the application to track the completion of the operation. The handle is returned back to the client by the "event handler callback" function registered with the driver.

The handle assigned to a client request expires when the client has been notified of the completion of the operation (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SST25VF064C_BLOCK_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_sst25vf064c.h

C

```
typedef enum {
  DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR
} DRV_SST25VF064C_BLOCK_EVENT;
```

Members

Members	Description
DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE	Block operation has been completed successfully. Read/Write/Erase Complete
DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR	There was an error during the block operation Read/Write/Erase Error

Description

SST25VF064C SPI Flash Driver Events

This enumeration identifies the possible events that can result from a Read, Write, or Erase request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SST25VF064C_BlockEventHandlerSet function when a block request is

completed.

DRV_SST25VF064C_CLIENT_STATUS Enumeration

Defines the client status.

File

drv_sst25vf064c.h

C

```
typedef enum {
  DRV_SST25VF064C_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0,
  DRV_SST25VF064C_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  DRV_SST25VF064C_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED,
  DRV_SST25VF064C_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR
} DRV_SST25VF064C_CLIENT_STATUS;
```

Members

Members	Description
DRV_SST25VF064C_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0	Up and running, ready to start new operations
DRV_SST25VF064C_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	Operation in progress, unable to start a new one
DRV_SST25VF064C_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED	Client is closed
DRV_SST25VF064C_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR	Client Error

Description

SPI Flash Client Status

Defines the various client status codes.

Remarks

None.

DRV_SST25VF064C_COMMAND_STATUS Enumeration

Specifies the status of the command for the read, write and erase operations.

File

drv_sst25vf064c.h

C

```
typedef enum {
  DRV_SST25VF064C_COMMAND_COMPLETED,
  DRV_SST25VF064C_COMMAND_QUEUED,
  DRV_SST25VF064C_COMMAND_IN_PROGRESS,
  DRV_SST25VF064C_COMMAND_ERROR_UNKNOWN
} DRV_SST25VF064C_COMMAND_STATUS;
```

Members

Members	Description
DRV_SST25VF064C_COMMAND_COMPLETED	Requested operation is completed
DRV_SST25VF064C_COMMAND_QUEUED	Scheduled but not started

DRV_SST25VF064C_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_SST25VF064C_COMMAND_ERROR_UNKNOWN	Unknown Command

Description

SST Flash Driver Command Status

SST Flash Driver command Status. This type specifies the status of the command for the read, write and erase operations.

Remarks

None.

DRV_SST25VF064C_EVENT_HANDLER Type

Pointer to a SST25VF064C SPI Flash Driver Event handler function.

File

drv_sst25vf064c.h

C

```
typedef void (* DRV_SST25VF064C_EVENT_HANDLER)(DRV_SST25VF064C_BLOCK_EVENT event,
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t context);
```

Returns

None.

Description

SST25VF064C SPI Flash Driver Event Handler Function Pointer

This data type defines the required function signature for the SST25VF064C SPI Flash driver event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values and return value are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE, it means that the data was transferred successfully. If the event is DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR, it means that the data was not transferred successfully. The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_SST25VF064C_BlockEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request. The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function. The Read, Write, and Erase functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running.

Example

```
void APP_MyBufferEventHandler
(
    DRV_SST25VF064C_BLOCK_EVENT event,
    DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
```

```
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write/Erase requests
context	Value identifying the context of the application that registered the event handling function

DRV_SST25VF064C_INIT Structure

Contains all the data necessary to initialize the SPI Flash device.

File

drv_sst25vf064c.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX spiDriverModuleIndex;
  PORTS_CHANNEL holdPortChannel;
  PORTS_BIT_POS holdBitPosition;
  PORTS_CHANNEL writeProtectPortChannel;
  PORTS_BIT_POS writeProtectBitPosition;
  PORTS_CHANNEL chipSelectPortChannel;
  PORTS_BIT_POS chipSelectBitPosition;
  uint32_t queueSize;
} DRV_SST25VF064C_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX spiDriverModuleIndex;	Identifies the SPI driver to be used
PORTS_CHANNEL holdPortChannel;	HOLD pin port channel
PORTS_BIT_POS holdBitPosition;	HOLD pin port position
PORTS_CHANNEL writeProtectPortChannel;	Write protect pin port channel
PORTS_BIT_POS writeProtectBitPosition;	Write Protect Bit pin position
PORTS_CHANNEL chipSelectPortChannel;	Chip select pin port channel
PORTS_BIT_POS chipSelectBitPosition;	Chip Select Bit pin position

uint32_t queueSize;	This is the buffer queue size. This is the maximum number of requests that this instance of the driver will queue. For a static build of the driver, this is overridden by the DRV_SST25VF064C_QUEUE_SIZE macro in system_config.h

Description

SST SPI Flash Driver Initialization Data

This structure contains all of the data necessary to initialize the SPI Flash device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_SST25VF064C_Initialize function.

DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID Macro

This value defines the SPI Flash Driver Block Command Invalid handle.

File

drv_sst25vf064c.h

C

```
#define DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID
```

Description

SPI Flash Driver Block Event Invalid Handle

This value defines the SPI Flash Driver Block Command Invalid handle. It is returned by read/write/erase routines when the request could not be taken.

Remarks

None.

DRV_SST25VF064C_INDEX_0 Macro

SPI Flash driver index definitions.

File

drv_sst25vf064c.h

C

```
#define DRV_SST25VF064C_INDEX_0 0
```

Description

Driver SPI Flash Module Index reference

These constants provide SST25VF064C SPI Flash driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_SST25VF064C_Initialize and DRV_SST25VF064C_Open routines to identify the driver instance in use.

DRV_SST25VF064C_INDEX_1 Macro

File

drv_sst25vf064c.h

C

```
#define DRV_SST25VF064C_INDEX_1 1
```

Description

This is macro DRV_SST25VF064C_INDEX_1.

Files

Files

Name	Description
drv_sst25vf016b.h	SPI Flash Driver Interface Definition
drv_sst25vf016b_config_template.h	SST25VF016B Driver Configuration Template.
drv_sst25vf020b.h	SPI Flash Driver Interface Definition
drv_sst25vf020b_config_template.h	SST25VF020B Driver Configuration Template.
drv_sst25vf064c.h	SPI Flash Driver Interface Definition
drv_sst25vf064c_config_template.h	SST25VF064C Driver Configuration Template.

Description

This section lists the source and header files used by the SPI Flash Driver Library.

drv_sst25vf016b.h

SPI Flash Driver Interface Definition

Enumerations

	Name	Description
	DRV_SST25VF016B_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF016B_CLIENT_STATUS	Defines the client status. **Implementation:** Dynamic

Functions

	Name	Description
	DRV_SST25VF016B_BlockErase	Erase the specified number of blocks in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockRead	Reads blocks of data starting from the specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockWrite	Write blocks of data starting from a specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF016B_ClientStatus	Gets current client-specific status of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF016B_Close	Closes an opened-instance of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF016B_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module. **Implementation:** Dynamic
	DRV_SST25VF016B_GeometryGet	Returns the geometry of the device. **Implementation:** Dynamic
	DRV_SST25VF016B_Initialize	Initializes the SST25VF016B SPI Flash Driver instance for the specified driver index. **Implementation:** Dynamic

	DRV_SST25VF016B_MediaIsAttached	Returns the status of the media. **Implementation:** Dynamic
	DRV_SST25VF016B_Open	Opens the specified SPI Flash driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SST25VF016B_Status	Gets the current status of the SPI Flash Driver module. **Implementation:** Dynamic
	DRV_SST25VF016B_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF016B_INDEX_0	SPI Flash driver index definitions
	DRV_SST25VF016B_INDEX_1	This is macro DRV_SST25VF016B_INDEX_1.

Structures

	Name	Description
	DRV_SST25VF016B_INIT	Contains all the data necessary to initialize the SPI Flash device. **Implementation:** Dynamic

Types

	Name	Description
	DRV_SST25VF016B_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_SST25VF016B_EVENT_HANDLER	Pointer to a SST25VF016B SPI Flash Driver Event handler function. **Implementation:** Dynamic

Description

SPI Flash Driver Interface Definition

The SPI Flash device driver provides a simple interface to manage the SPI Flash modules which are external to Microchip Controllers. This file defines the interface definition for the SPI Flash Driver.

File Name

drv_sst25vf016b.h

Company

Microchip Technology Inc.

drv_sst25vf016b_config_template.h

SST25VF016B Driver Configuration Template.

Macros

	Name	Description
	DRV_SST25VF016B_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.

	DRV_SST25VF016B_HARDWARE_HOLD_ENABLE	Specifies if the hardware hold feature is enabled or not.
	DRV_SST25VF016B_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies if the hardware write protect feature is enabled or not.
	DRV_SST25VF016B_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_SST25VF016B_MODE	Determines whether the driver is implemented as static or dynamic
	DRV_SST25VF016B_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

Description

SST25VF016B Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_sst25vf016b_config_template.h

Company

Microchip Technology Inc.

drv_sst25vf020b.h

SPI Flash Driver Interface Definition

Enumerations

	Name	Description
	DRV_SST25VF020B_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF020B_CLIENT_STATUS	Defines the client status.
	DRV_SST25VF020B_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.

Functions

	Name	Description
	DRV_SST25VF020B_BlockErase	Erase the specified number of blocks in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockEraseWrite	Erase and Write blocks of data starting from a specified address in SST flash memory.
	DRV_SST25VF020B_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockRead	Reads blocks of data starting from the specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockWrite	Write blocks of data starting from a specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_ClientStatus	Gets current client-specific status of the SPI Flash driver. **Implementation:** Dynamic

	DRV_SST25VF020B_Close	Closes an opened-instance of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF020B_CommandStatus	Gets the current status of the command.
	DRV_SST25VF020B_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module. **Implementation:** Dynamic
	DRV_SST25VF020B_GeometryGet	Returns the geometry of the device. **Implementation:** Dynamic
	DRV_SST25VF020B_Initialize	Initializes the SST25VF020B SPI Flash Driver instance for the specified driver index. **Implementation:** Dynamic
	DRV_SST25VF020B_MediaIsAttached	Returns the status of the media. **Implementation:** Dynamic
	DRV_SST25VF020B_Open	Opens the specified SPI Flash driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SST25VF020B_Status	Gets the current status of the SPI Flash Driver module. **Implementation:** Dynamic
	DRV_SST25VF020B_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF020B_INDEX_0	SPI Flash driver index definitions.
	DRV_SST25VF020B_INDEX_1	This is macro DRV_SST25VF020B_INDEX_1.

Structures

	Name	Description
	DRV_SST25VF020B_INIT	Contains all the data necessary to initialize the SPI Flash device.

Types

	Name	Description
	DRV_SST25VF020B_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_SST25VF020B_EVENT_HANDLER	Pointer to a SST25VF020B SPI Flash Driver Event handler function.

Description

SPI Flash Driver Interface Definition

The SPI Flash device driver provides a simple interface to manage the SPI Flash modules which are external to Microchip Controllers. This file defines the interface definition for the SPI Flash Driver.

File Name

drv_sst25vf020b.h

Company

Microchip Technology Inc.

drv_sst25vf020b_config_template.h

SST25VF020B Driver Configuration Template.

Macros

	Name	Description
	DRV_SST25VF020B_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_SST25VF020B_HARDWARE_HOLD_ENABLE	Specifies if the hardware hold feature is enabled or not.
	DRV_SST25VF020B_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies if the hardware write protect feature is enabled or not.
	DRV_SST25VF020B_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_SST25VF020B_MODE	Determines whether the driver is implemented as static or dynamic.
	DRV_SST25VF020B_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

Description

SST25VF020B Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_sst25vf020b_config_template.h

Company

Microchip Technology Inc.

drv_sst25vf064c.h

SPI Flash Driver Interface Definition

Enumerations

	Name	Description
	DRV_SST25VF064C_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF064C_CLIENT_STATUS	Defines the client status.
	DRV_SST25VF064C_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.

Functions

	Name	Description
	DRV_SST25VF064C_BlockErase	Erase the specified number of blocks in Flash memory.
	DRV_SST25VF064C_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

	DRV_SST25VF064C_BlockRead	Reads blocks of data starting from the specified address in Flash memory.
	DRV_SST25VF064C_BlockWrite	Write blocks of data starting from a specified address in Flash memory.
	DRV_SST25VF064C_ClientStatus	Gets current client-specific status of the SPI Flash driver.
	DRV_SST25VF064C_Close	Closes an opened-instance of the SPI Flash driver.
	DRV_SST25VF064C_CommandStatus	Gets the current status of the command.
	DRV_SST25VF064C_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module.
	DRV_SST25VF064C_GeometryGet	Returns the geometry of the device.
	DRV_SST25VF064C_Initialize	Initializes the SST25VF064C SPI Flash Driver instance for the specified driver index.
	DRV_SST25VF064C_MediaIsAttached	Returns the status of the media.
	DRV_SST25VF064C_Open	Opens the specified SPI Flash driver instance and returns a handle to it.
	DRV_SST25VF064C_Status	Gets the current status of the SPI Flash Driver module.
	DRV_SST25VF064C_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR.

Macros

	Name	Description
	DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF064C_INDEX_0	SPI Flash driver index definitions.
	DRV_SST25VF064C_INDEX_1	This is macro DRV_SST25VF064C_INDEX_1.

Structures

	Name	Description
	DRV_SST25VF064C_INIT	Contains all the data necessary to initialize the SPI Flash device.

Types

	Name	Description
	DRV_SST25VF064C_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_SST25VF064C_EVENT_HANDLER	Pointer to a SST25VF064C SPI Flash Driver Event handler function.

Description

SPI Flash Driver Interface Definition

The SPI Flash device driver provides a simple interface to manage the SPI Flash modules which are external to Microchip Controllers. This file defines the interface definition for the SPI Flash Driver.

File Name

drv_sst25vf064c.h

Company

Microchip Technology Inc.

drv_sst25vf064c_config_template.h

SST25VF064C Driver Configuration Template.

Macros

	Name	Description
	DRV_SST25VF064C_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_SST25VF064C_HARDWARE_HOLD_ENABLE	Specifies whether or not the hardware hold feature is enabled.
	DRV_SST25VF064C_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies if whether or not the hardware write protect feature is enabled.
	DRV_SST25VF064C_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_SST25VF064C_MODE	Determines whether the driver is implemented as static or dynamic.
	DRV_SST25VF064C_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

Description

SST25VF064C Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_sst25vf064c_config_template.h

Company

Microchip Technology Inc.

Timer Driver Library

This topic describes the Timer Driver Library.

Introduction

This library provides an interface to manage the Timer module on the Microchip family of microcontrollers during different modes of operation.

Description

Timers are useful for generating accurate time based periodic interrupts for software application or real time operating systems. Other uses include counting external pulses or accurate timing measurement of external events using the timer's gate functions and accurate hardware delays.

Note: Not all features are available on all devices. Please refer to the specific device data sheet to determine availability.

Using the Library

This topic describes the basic architecture of the Timer Driver Library and provides information and examples on its use.

Description

Interface Header File: drv_tmr.h

The interface to the Timer Driver Library is defined in the drv_tmr.h header file.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

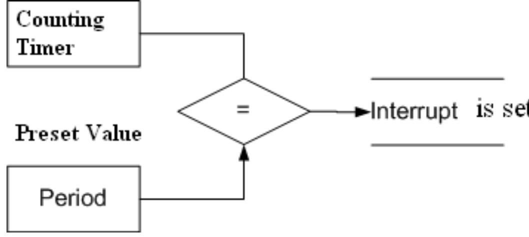
Abstraction Model

The Timer Driver abstracts the hardware by providing the capability to register callback functions to the application.

Description

Abstraction Model

The abstraction model of the Timer Driver is explained in the following diagram:



The core functionality of the Timer allows access to both the counter and the period values.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Timer Driver Library.

Library Interface Section	Description
Configuration	Provides macros for configuring the system. It is required that the system configures the driver to build correctly by choosing appropriate configuration options as listed in this section. These macros enable different features or modes of the timer peripheral.
System Interaction Functions	Provides interfaces to system layer to initialize, deinitialize and reinitialize the module. This section also describes functions to query the status of the module.
Core Functions	Provides interfaces for core functionality of the driver.
Alarm Functions	Provides interfaces to handle alarm features, if alarm functionality is enabled.
Period Functions	Provides interfaces to control the periodicity of the timers.
Counter Control Functions	Provides interfaces to update the counter values.

Miscellaneous Functions	Provides interfaces to get the version information, timer tick and operating frequencies.

How the Library Works

The library provides interfaces to support:

- System Interaction
- Sync Mode Setup
- Period Modification
- Counter Modification
- Client Core Functionality
- Client Alarm Functionality (optional function, enabled using configuration options)
- Other Optional Functionality (enabled using configuration options)

Note: Any code segment pertaining to the driver interfaces will work for both the static or dynamic configurations. It is not necessary to modify the code to move from one configuration to the other (i.e., from static or dynamic or static-multi).

System Interaction

This section describes Timer initialization and reinitialization.

Description

Initialization and Reinitialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized.

The DRV_TMR_Initialize function returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_TMR_Deinitialize, DRV_TMR_Status, DRV_TMR_Tasks, and DRV_TMR_Tasks_ISR.

Example: Timer Initialization

```
DRV_TMR_INIT      init;
SYS_MODULE_OBJ    object;
SYS_STATUS        tmrStatus;

// populate the TMR init configuration structure
init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
init.tmrId             = TMR_ID_2;
init.clockSource       = TMR_CLOCK_SOURCE_PERIPHERAL_CLOCK;
init.prescale          = TMR_PRESCALE_VALUE_256;
init.interruptSource   = INT_SOURCE_TIMER_2;
init.mode              = DRV_TMR_OPERATION_MODE_16_BIT;
init.asyncWriteEnable = false;

object = DRV_TMR_Initialize (DRV_TMR_INDEX_0, (SYS_MODULE_INIT *)&init);

if (object == SYS_MODULE_OBJ_INVALID)
{
    // Handle error
}
```

Deinitialization

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This routine may block if the driver is running in an OS environment that supports blocking operations and the driver requires system resources access. However, the routine will never block for hardware Timer access.

Status

Timer status is available to query the module state after initialization and reinitialization.

Tasks Routine

The interface DRV_TMR_Tasks needs to be called by the system task service in a polled environment and the interface DRV_TMR_Tasks_ISR needs to be called by the ISR of the timer in an interrupt-based system.

Example: Polling

```
int main( void )
{
    SYS_MODULE_OBJ object;
    object = DRV_TMR_Initialize( DRV_TMR_INDEX_0, (SYS_MODULE_INIT *) &initConf );

    if( SYS_STATUS_READY != DRV_TMR_Status( object ) )
        return 0;

    while (1)
    {
        DRV_TMR_Tasks (object);
    }
}
```

Example: Interrupt

```
int main( void )
```

```
{
    SYS_MODULE_OBJ object;
    object = DRV_TMR_Initialize( DRV_TMR_INDEX_0, (SYS_MODULE_INIT *) &initConf );

    if( SYS_STATUS_READY != DRV_TMR_Status( object ) )
        return 0;

    while (1);
}

/* Sample interrupt routine not specific to any device family */
void ISR T1Interrupt(void)
{
    //Call the Timer Tasks routine
    DRV_TMR_Tasks_ISR(object);
}
```

Client Interaction

This section describes general client operation.

Description

General Client Operation

For the application to begin using an instance of the Timer module, it must call the DRV_TMR_Open function. This provides the configuration required to open the Timer instance for operation.

The Timer Driver supports only the 'DRV_IO_INTENT_EXCLUSIVE' IO_INTENT.

Example:

```
DRV_HANDLE handle;

// Configure the instance DRV_TMR_INDEX_1 with the configuration
handle = DRV_TMR_Open(DRV_TMR_INDEX_1, DRV_IO_INTENT_EXCLUSIVE);

if( handle == DRV_HANDLE_INVALID )
{
    // Client cannot open the instance.
}
```

The function DRV_TMR_Close closes an already opened instance of the Timer Driver, invalidating the handle. DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example:

```
DRV_HANDLE handle;

// Configure the instance DRV_TMR_INDEX_1 with the configuration
handle = DRV_TMR_Open(DRV_TMR_INDEX_1, DRV_IO_INTENT_EXCLUSIVE);

/*...*/

DRV_TMR_Close( handle );
```

The client has the option to check the status through the interface DRV_TMR_ClientStatus.

Example:

```
DRV_HANDLE handle;

// Configure the instance DRV_TMR_INDEX_1 with the configuration
handle = DRV_TMR_Open(DRV_TMR_INDEX_1, DRV_IO_INTENT_EXCLUSIVE);

if ( DRV_TMR_CLIENT_STATUS_READY != DRV_TMR_ClientStatus( handle ) )
   return 0;
```

Period Modification

This section describes Period modification for the different types of Timers (i.e., 16-/32-bit).

Description

These set of functions help modify the Timer periodicity at the client level, regardless of whether it is an overflow or a period match-based Timer.

This interface can be used to alter the already set periodicity at the system level interface DRV_TMR_Initialize.

Period Modification

Periodicity for the type of Timer (16-/32-bit) can be controlled as follows:

- Timer periodicity can be modified using DRV_TMR_AlarmPeriodSet and the current period value can be obtained using DRV_TMR_AlarmPeriodGet
- 16-bit timer periodicity can be modified using DRV_TMR_AlarmPeriod16BitSet and the current period value can be obtained using DRV_TMR_Period16BitGet
- 32-bit timer periodicity can be modified using DRV_TMR_AlarmPeriod32BitSet and the current period value can be obtained using DRV_TMR_AlarmPeriod32BitGet

Example:

```
DRV_HANDLE handle;
/* Open the client */
handle = DRV_TMR_Open( DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );

/* ... */

/* Update the new period */
DRV_TMR_AlarmPeriod16BitSet( handle, 0xC350);
```

Counter Modification

This section describes counter modification for the different types of Timers (i.e., 8-/16-/32-bit).

Description

These set of functions help modify the initial value of the Timer counters to help adjust any errors in the periodicity.

Counter Modification

Initial counter values can be controlled as follows:

- Timer initial value can be modified using DRV_TMR_CounterValueSet and the current counter value can be obtained using DRV_TMR_CounterValueGet
- 16-bit timer initial value can be modified using DRV_TMR_CounterValue16BitSet and the current counter value can be obtained using DRV_TMR_CounterValue16BitGet
- 32-bit timer initial value can be modified using DRV_TMR_CounterValue32BitSet and the current counter value can be obtained using DRV_TMR_CounterValue32BitGet

Example:

```
DRV_HANDLE handle;
/* Open the client */
handle = DRV_TMR_Open( DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );

/* ... */

/* Update the counter value */
/* Following code updates the initial value from 0x0000 to 0x0010
   to cover up any error in the previously set periodicity */

DRV_TMR_CounterValue16BitSet( handle, 0x0010);
```

Core Functionality

This section describes core functionality of the Timer Driver.

Description

Core functionality provides an extremely basic interface for the driver operation.

Applications using the Timer core functionality need to perform the following:

1. The system should have completed the necessary initialization and DRV_TMR_Tasks or DRV_TMR_Tasks_ISR should called in a polled/interrupt environment.
2. Open_the driver using DRV_TMR_Open. The Timer Driver only supports exclusive access.
3. The Timer period can be updated using DRV_TMR_AlarmPeriodSet (or 16-bit/32-bit versions) if the client intends to override the already preset value during the initialization. The previously set value can be retrieved using DRV_TMR_AlarmPeriodGet.
4. Start the driver using DRV_TMR_Start.
5. Poll for the elapsed alarm status using DRV_TMR_AlarmHasElapsed.
6. The client will be able to stop the started Timer instance using DRV_TMR_Stop at any time and will be able to close it using DRV_TMR_Close when it is no longer required.

Example:

```
/* Open the client */
handle = DRV_TMR_Open( DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );
DRV_TMR_Start (handle);
unsigned int alarmCount = 0;
while (1)
{
    if (true == DRV_TMR_AlarmHasElapsed (handle))
    {
        alarmCount++;
        // Do something
    }
}
```

Notes:

1. The user needs to stop the Timer before any updates on the counter or period and restart it later.
2. The Timer alarm count gets reset after any call to DRV_TMR_AlarmHasElapsed.
3. The Timer alarm status remains unchanged if the user stops the timer and restarts later.

© 2013-2015 Microchip Technology Inc.

Alarm Functionality

This section describes the Timer Driver alarm functionality.

Description

The Timer Driver provides alarm functionality.

Applications using the Timer alarm functionality, need to perform the following:

1. The system should have completed the necessary initialization and DRV_TMR_Tasks/DRV_TMR_Tasks_ISR should be running in either a polled environment or in an interrupt environment.
2. Open the driver using DRV_TMR_Open. The Timer Driver supports exclusive access only.
3. Configure the alarm using DRV_TMR_AlarmRegister.
4. Start the driver using DRV_TMR_Start.
5. If a callback is supplied, the Timer Driver will call the callback function when the alarm expires.
6. The client will be able to stop the started Timer module instance using DRV_TMR_Stop at any time and will be able to close it using DRV_TMR_Close when it is no longer required.
7. The client can deregister the callback by using DRV_TMR_AlarmDeregister.

Example:

```
DRV_HANDLE handle;
/* Open the client */
handle = DRV_TMR_Open (DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
/* Configure the timer alarm feature */
uint32_t myFreq = 1000; // 1KHz
uint32_t clkFreq = DRV_TMR_CounterFrequencyGet(tmrHandle);   // timer running frequency

// calculate the divider needed
uint32_t divider = clkFreq / myFreq;

// Start the alarm
if(!DRV_TMR_AlarmRegister ( tmrHandle, divider, true, 0, CallBackFreq ))
{
    // divider value could not be obtain;
    // handle the error
    //
    return;
}

DRV_TMR_Start (handle);

// The driver tasks function calls the client registered callback after the alarm expires.
void CallBackFreq (uintptr_t context, uint32_t alarmCount)
{
    // Do something specific on an alarm event trigger
}
```

Optional Interfaces

This section describes additional/optional client interfaces.

Description

Additional/Optional client interfaces include the following:

Get Operating Frequency

The function DRV_TMR_CounterFrequencyGet provides the client with the information on the Timer operating frequency.

Example:

```
DRV_HANDLE handle;
uint32_t freq;

/* Open the client */
handle = DRV_TMR_Open (DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);

freq = DRV_TMR_OperatingFrequencyGet (handle);
```

Example Usage of the Timer Driver

This section describes typical usage of the Timer Driver for various Timer modules in polling/interrupt advanced/core modes.

Description

The user can pass NULL to the driver initialize interface. However, the respective configuration parameters need to be configured in the correct manner.

Example:

```
//Polled mode under 32-bit count mode for a PIC32 device using the alarm feature
SYS_MODULE_OBJ object;

// main
DRV_TMR_INIT init;
DRV_HANDLE handle;

init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
init.tmrId            = TMR_ID_2;
init.clockSource      = TMR_CLKSOURCE_INTERNAL;
init.prescale         = TMR_PRESCALE_TX_VALUE_256;
init.interruptSource  = INT_SOURCE_TIMER_3;
init.timerPeriod      = 0xFF00;

object = DRV_TMR_Initialize (DRV_TMR_INDEX_0, (SYS_MODULE_INIT *)&init);
if ( SYS_STATUS_READY != DRV_TMR_Status(object))
    return 0;

handle = DRV_TMR_Open (DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if ( DRV_TMR_STATUS_READY != DRV_TMR_ClientStatus(handle))
    return 0;

if(!DRV_TMR_AlarmRegister ( tmrHandle, divider, true, 0, AlarmCallback ))
{
    // divider value could not be obtain;
    // handle the error
}

DRV_TMR_Start (handle);

while (1)
{
    DRV_TMR_Tasks (object);
}

DRV_TMR_Stop (handle);

DRV_TMR_Close (handle);
if ( DRV_TMR_STATUS_INVALID != DRV_TMR_ClientStatus(handle))
    return 0;

DRV_TMR_Deinitialize (object);
// end main

void AlarmCallback (uintptr_t context, uint32_t alarmCount)
{
  // Do something specific on an alarm event trigger
}
```

Configuring the Library

Macros

	Name	Description
	DRV_TMR_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_TMR_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_TMR_CLOCK_PRESCALER	Sets the default timer driver clock prescaler.
	DRV_TMR_MODE	Sets the default timer driver clock operating mode.
	DRV_TMR_MODULE_ID	Sets the default timer module ID to be used by the timer driver.
	DRV_TMR_MODULE_INIT	Sets the default module init value for the timer driver.
	DRV_TMR_INTERRUPT_SOURCE	Sets the default timer driver clock interrupt source
	DRV_TMR_ASYNC_WRITE_ENABLE	Controls Asynchronous Write mode of the Timer.
	DRV_TMR_CLOCK_SOURCE	Sets the default timer driver clock source.
	DRV_TMR_CLIENTS_NUMBER	Sets up the maximum number of clients that can be supported by an instance of the dynamic driver.

Description

The configuration of the Timer Driver Library is based on the file `system_config.h`.

This header file contains the configuration selection for the Timer Driver Library build. Based on the selections made here and the system setup, the Timer Driver may support the selected features. These configuration settings will apply to all instances of the driver.

This header can be placed anywhere in the application-specific folders and the path of this header needs to be presented to the include search for a successful build. Refer to the Applications Overview section for more details.

DRV_TMR_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported by the dynamic driver.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_INSTANCES_NUMBER 5
```

Description

Hardware instances support

This definition sets up the maximum number of hardware instances that can be supported by the dynamic driver.

Remarks

None

DRV_TMR_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_INTERRUPT_MODE true
```

Description

TMR Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of timer operation is desired
- false - Select if polling mode of timer operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

DRV_TMR_CLOCK_PRESCALER Macro

Sets the default timer driver clock prescaler.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_CLOCK_PRESCALER (TMR_PRESCALE_VALUE_256)
```

Description

Default timer driver clock prescaler

This macro sets the default timer driver clock prescaler.

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_MODE Macro

Sets the default timer driver clock operating mode.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_MODE (DRV_TMR_OPERATION_MODE_16_BIT)
```

Description

Default timer driver clock operating mode

This macro sets the default timer driver clock operating mode.

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_MODULE_ID Macro

Sets the default timer module ID to be used by the timer driver.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_MODULE_ID (TMR_ID_2)
```

Description

Default timer driver index

This macro sets the default timer module ID to be used by the timer driver.

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_MODULE_INIT Macro

Sets the default module init value for the timer driver.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_MODULE_INIT (SYS_MODULE_POWER_RUN_FULL)
```

Description

Default module init object configuration

This macro sets the default module init value for the timer driver.

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_INTERRUPT_SOURCE Macro

Sets the default timer driver clock interrupt source

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_INTERRUPT_SOURCE (INT_SOURCE_TIMER_2)
```

Description

Default timer driver clock interrupt source

This macro sets the default timer driver clock interrupt source

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_ASYNC_WRITE_ENABLE Macro

Controls Asynchronous Write mode of the Timer.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_ASYNC_WRITE_ENABLE false
```

Description

TMR Asynchronous write mode configuration

This macro controls the Asynchronous Write mode of the Timer. This macro accepts the following values:

- true - Configures the Timer to enable asynchronous write control
- false - Configures the Timer to disable asynchronous write control
- DRV_CONFIG_NOT_SUPPORTED - When the feature is not supported on the instance.

Remarks

This feature is not available in all modules/devices. Refer to the specific device data sheet for more information.

DRV_TMR_CLOCK_SOURCE Macro

Sets the default timer driver clock source.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_CLOCK_SOURCE (DRV_TMR_CLKSOURCE_INTERNAL)
```

Description

Default timer driver clock source

This macro sets the default timer driver clock source.

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be supported by an instance of the dynamic driver.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_CLIENTS_NUMBER 1
```

Description

Client instances support

This definition sets up the maximum number of clients that can be supported by an instance of the dynamic driver.

Remarks

Currently each client is required to get exclusive access to the timer module. Therfore the DRV_TMR_CLIENTS_NUMBER should always be set to 1.

Building the Library

This section lists the files that are available in the Timer Driver Library.

Description

This section list the files that are available in the `\src` folder of the Timer Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/tmr`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_tmr.h`	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_tmr_dynamic.c`	Basic Timer driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The Timer Driver Library depends on the following modules:

- Clock System Service Library
- Interrupt System Service Library
- Interrupt Peripheral Library
- Device Control System Service Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_TMR_Deinitialize	Deinitializes the specified instance of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_Initialize	Initializes the Timer driver. **Implementation:** Static/Dynamic
	DRV_TMR_Status	Provides the current status of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_Tasks	Maintains the driver's state machine. **Implementation:** Dynamic
	DRV_TMR_ClockSet	Sets the timers clock by selecting the source and prescaler. **Implementation:** Dynamic
	DRV_TMR_GateModeSet	Enables the Gate mode. **Implementation:** Dynamic

b) Core Functions

	Name	Description
	DRV_TMR_ClientStatus	Gets the status of the client operation. **Implementation:** Dynamic
	DRV_TMR_Close	Closes an opened instance of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_Open	Opens the specified Timer driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_TMR_Start	Starts the Timer counting. **Implementation:** Static/Dynamic
	DRV_TMR_Stop	Stops the Timer from counting. **Implementation:** Static/Dynamic

c) Alarm Functions

	Name	Description
	DRV_TMR_Alarm16BitDeregister	Removes a previously set alarm. **Implementation:** Dynamic
	DRV_TMR_Alarm32BitDeregister	Removes a previously set alarm. **Implementation:** Dynamic
	DRV_TMR_AlarmHasElapsed	Provides the status of Timer's period elapse. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriod16BitGet	Provides the 16-bit Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriod16BitSet	Updates the 16-bit Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriod32BitGet	Provides the 32-bit Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriod32BitSet	Updates the 32-bit Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmDisable	Disables an alarm signal. **Implementation:** Dynamic

	DRV_TMR_AlarmEnable	Re-enables an alarm signal. **Implementation:** Dynamic
	DRV_TMR_Alarm16BitRegister	Sets up an alarm. **Implementation:** Dynamic
	DRV_TMR_Alarm32BitRegister	Sets up an alarm. **Implementation:** Dynamic
	DRV_TMR_AlarmDeregister	Removes a previously set alarm. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriodGet	Provides the Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriodSet	Updates the Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmRegister	Sets up an alarm. **Implementation:** Dynamic

d) Counter Control Functions

	Name	Description
	DRV_TMR_CounterFrequencyGet	Provides the Timer input frequency. **Implementation:** Dynamic
	DRV_TMR_CounterValue16BitGet	Reads the 16-bit Timer's counter register. **Implementation:** Dynamic
	DRV_TMR_CounterValue16BitSet	Updates the 16-bit Timer's counter register. **Implementation:** Dynamic
	DRV_TMR_CounterValue32BitGet	Reads the 32-bit Timer's counter register. **Implementation:** Dynamic
	DRV_TMR_CounterValue32BitSet	Updates the 32-bit Timer's counter register. **Implementation:** Dynamic
	DRV_TMR_CounterClear	Clears the Timer's counter register. **Implementation:** Static/Dynamic
	DRV_TMR_CounterValueGet	Reads the Timer's counter register. **Implementation:** Static/Dynamic
	DRV_TMR_CounterValueSet	Updates the Timer's counter register. **Implementation:** Static/Dynamic

e) Miscellaneous Functions

	Name	Description
	DRV_TMR_GateModeClear	Enables the Gate mode. **Implementation:** Dynamic
	DRV_TMR_PrescalerGet	This function gets the currently selected prescaler. **Implementation:** Dynamic
	DRV_TMR_OperationModeGet	This function gets the currently selected operation mode. **Implementation:** Dynamic
	DRV_TMR_DividerRangeGet	Returns the Timer divider values. **Implementation:** Dynamic
	DRV_TMR_Tasks_ISR	Maintains the driver's state machine, processes the events and implements its ISR. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	DRV_TMR_CALLBACK	Pointer to a Timer driver callback function data type.
	DRV_TMR_INIT	Defines the Timer driver initialization data.
	DRV_TMR_CLIENT_STATUS	Identifies the client-specific status of the Timer driver
	DRV_TMR_CLK_SOURCES	Lists the clock sources available for timer driver.
	DRV_TMR_DIVIDER_RANGE	This data structure specifies the divider values that can be obtained by the timer module.
	DRV_TMR_OPERATION_MODE	Lists the operation modes available for timer driver.
	DRV_TMR_INDEX_COUNT	Number of valid Timer driver indices.
	DRV_TMR_INDEX_0	Timer driver index definitions
	DRV_TMR_INDEX_1	This is macro DRV_TMR_INDEX_1.
	DRV_TMR_INDEX_2	This is macro DRV_TMR_INDEX_2.
	DRV_TMR_INDEX_3	This is macro DRV_TMR_INDEX_3.
	DRV_TMR_INDEX_4	This is macro DRV_TMR_INDEX_4.
	DRV_TMR_INDEX_5	This is macro DRV_TMR_INDEX_5.
	DRV_TMR_INDEX_6	This is macro DRV_TMR_INDEX_6.
	DRV_TMR_INDEX_7	This is macro DRV_TMR_INDEX_7.
	DRV_TMR_INDEX_8	This is macro DRV_TMR_INDEX_8.
	DRV_TMR_INDEX_9	This is macro DRV_TMR_INDEX_9.

Description

This section describes the functions of the Timer Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_TMR_Deinitialize Function

Deinitializes the specified instance of the Timer driver.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the Timer driver, disabling its operation (and any hardware). All internal data is invalidated.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_TMR_Status operation. The system has to use DRV_TMR_Status to find out when the module is in the ready state.

Preconditions

The DRV_TMR_Initialize function must have been called before calling this function and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      tmrObject;      //  Returned from DRV_TMR_Initialize
SYS_STATUS          tmrStatus;

DRV_TMR_Deinitialize ( tmrObject );

tmrStatus = DRV_TMR_Status ( tmrObject );

if ( SYS_MODULE_UNINITIALIZED == tmrStatus )
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_TMR_Initialize

Function

void DRV_TMR_Deinitialize (SYS_MODULE_OBJ object)

DRV_TMR_Initialize Function

Initializes the Timer driver.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
SYS_MODULE_OBJ DRV_TMR_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *
const init);
```

Returns

If successful, returns a valid handle to a driver object. Otherwise, it returns SYS_MODULE_OBJ_INVALID. The returned object must be passed as argument to DRV_TMR_Deinitialize, DRV_TMR_Tasks, DRV_TMR_Tasks_ISR and DRV_TMR_Status functions.

Description

This function initializes the Timer driver, making it ready for clients to open and use it.

Remarks

This function must be called before any other Timer driver function is called.

This function should only be called once during system initialization unless DRV_TMR_Deinitialize is called to deinitialize the driver instance.

This function will NEVER block for hardware access. The system must use DRV_TMR_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
DRV_TMR_INIT    init;
SYS_MODULE_OBJ  objectHandle;

// Populate the timer initialization structure
init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
init.tmrId            = TMR_ID_2;
init.clockSource      = DRV_TMR_CLKSOURCE_INTERNAL;
init.prescale         = TMR_PRESCALE_VALUE_256;
init.interruptSource  = INT_SOURCE_TIMER_2;
init.mode             = DRV_TMR_OPERATION_MODE_16_BIT;
init.asyncWriteEnable = false;

// Do something

objectHandle = DRV_TMR_Initialize ( DRV_TMR_INDEX_0, (SYS_MODULE_INIT*)&init );

if ( SYS_MODULE_OBJ_INVALID == objectHandle )
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Index for the driver instance to be initialized

init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_TMR_Initialize (const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT * const init)

DRV_TMR_Status Function

Provides the current status of the Timer driver.

Implementation: Dynamic

File

drv_tmr.h

C

```
SYS_STATUS DRV_TMR_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is initialized and ready for operation

Description

This function provides the current status of the Timer driver.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_STATUS_ERROR - Indicates that the driver is in an error state

Any value less than SYS_STATUS_ERROR is also an error state.

SYS_MODULE_UNINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

The this operation can be used to determine when any of the driver's module level operations has completed.

Once the status operation returns SYS_STATUS_READY, the driver is ready for operation.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

Preconditions

The DRV_TMR_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_TMR_Initialize
SYS_STATUS          tmrStatus;

tmrStatus = DRV_TMR_Status ( object );

else if ( SYS_STATUS_ERROR >= tmrStatus )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_TMR_Initialize

Function

SYS_STATUS DRV_TMR_Status (SYS_MODULE_OBJ object)

DRV_TMR_Tasks Function

Maintains the driver's state machine.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Tasks(SYS_MODULE_OBJ object);
```

Returns

None

Description

This function is used to maintain the driver's internal state machine and processes the timer events in non interrupt-driven implementations (DRV_TMR_INTERRUPT_MODE == false).

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks)

Preconditions

The DRV_TMR_Initialize function must have been called for the specified Timer driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_TMR_Initialize

while (true)
{
    DRV_TMR_Tasks ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_TMR_Initialize)

Function

void DRV_TMR_Tasks (SYS_MODULE_OBJ object)

DRV_TMR_ClockSet Function

Sets the timers clock by selecting the source and prescaler.

Implementation: Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_ClockSet(DRV_HANDLE handle, DRV_TMR_CLK_SOURCES clockSource, TMR_PRESCALE
preScale);
```

Returns

- true - if the operation is successful
- false - either the handle is invalid or the clockSource and/or prescaler are not supported

Description

This function sets the timers clock by selecting the source and prescaler.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 32-Bit timer mode if mode selection is applicable.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE              tmrHandle;  // Returned from DRV_TMR_Open

DRV_TMR_ClockSet ( tmrHandle, DRV_TMR_CLKSOURCE_INTERNAL, TMR_PRESCALE_TX_VALUE_256 );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
clockSource	Clock source of the timer
preScale	Timer's Prescaler divisor

Function

bool DRV_TMR_ClockSet (DRV_HANDLE handle, DRV_TMR_CLK_SOURCES clockSource,
TMR_PRESCALE preScale)

DRV_TMR_GateModeSet Function

Enables the Gate mode.

Implementation: Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_GateModeSet(DRV_HANDLE handle);
```

Returns

- true - if the operation is successful
- false - either the handle is invalid or the gate mode is not supported

Description

This function enables the Gated mode of Timer. User can measure the duration of an external signal in this mode. Once the Gate mode is enabled, Timer will start on the raising edge of the external signal. It will keep counting until the next falling edge.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

DRV_TMR_GateModeSet ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

bool DRV_TMR_GateModeSet (DRV_HANDLE handle)

b) Core Functions

DRV_TMR_ClientStatus Function

Gets the status of the client operation.

Implementation: Dynamic

File

drv_tmr.h

C

```
DRV_TMR_CLIENT_STATUS DRV_TMR_ClientStatus(DRV_HANDLE handle);
```

Returns

None

Description

This function gets the status of the recently completed client level operation.

Preconditions

The DRV_TMR_Initialize function must have been called for the specified Timer driver instance.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
DRV_TMR_CLIENT_STATUS tmrDrvStatus;

tmrDrvStatus = DRV_TMR_ClientStatus ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_TMR_CLIENT_STATUS DRV_TMR_ClientStatus (DRV_HANDLE handle)

DRV_TMR_Close Function

Closes an opened instance of the Timer driver.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Close(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the Timer driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver functions. A new handle must be obtained by calling DRV_TMR_Open before the caller may use the driver again.

Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_TMR_Initialize function must have been called for the specified Timer driver instance.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TMR_Open

DRV_TMR_Close ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TMR_Close (DRV_HANDLE handle)

DRV_TMR_Open Function

Opens the specified Timer driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_tmr.h

C

```
DRV_HANDLE DRV_TMR_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

If successful, the function returns a valid open instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID.

Description

This function opens the specified Timer driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. Timer driver does not support multiple clients. If two tasks want to use the timer, one should wait until the other one gets closed.

Remarks

The handle returned is valid until the DRV_TMR_Close function is called.

This function will NEVER block waiting for hardware.

If the requested intent flags are not supported, the function will return DRV_HANDLE_INVALID.

The Timer driver does not support DRV_IO_INTENT_SHARED. Only exclusive access is supported for now.

Preconditions

The DRV_TMR_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_TMR_Open ( DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );

if ( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_TMR_Open (const SYS_MODULE_INDEX index,

const DRV_IO_INTENT intent)

DRV_TMR_Start Function

Starts the Timer counting.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_Start(DRV_HANDLE handle);
```

Returns

- true - if the operation succeeded
- false - the supplied handle is invalid or the client doesn't have the needed parameters to run (alarm callback and period)

Description

This function starts the Timer counting.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Timer parameters must have been set by a call to DRV_TMR_AlarmRegister.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

DRV_TMR_AlarmRegister(tmrHandle, 0x100, true, 0, myTmrCallback);
DRV_TMR_Start ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

bool DRV_TMR_Start (DRV_HANDLE handle)

DRV_TMR_Stop Function

Stops the Timer from counting.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Stop(DRV_HANDLE handle);
```

Returns

None.

Description

This function stops the running Timer from counting.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TMR_Open

DRV_TMR_Stop ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TMR_Stop (DRV_HANDLE handle)

c) Alarm Functions

DRV_TMR_Alarm16BitDeregister Function

Removes a previously set alarm.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Alarm16BitDeregister(DRV_HANDLE handle);
```

Returns

None.

Description

This function removes a previously set alarm. This API is valid only if the 16-bit mode of the timer is selected Otherwise use DRV_TMR_Alarm32BitDeregister function.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 16-Bit timer mode if mode selection is applicable.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

DRV_TMR_Alarm16BitRegister function must have been called before.

Example

```
//Example of a key debounce check

static unsigned int lastReadKey, readKey, keyCount, globalKeyState;
DRV_HANDLE              tmrHandle;  // Returned from DRV_TMR_Open
//Do the initialization with 'mode' set to DRV_TMR_OPERATION_MODE_16_BIT

void keyPressDetect ()
{
    // Calculate the count to be passed on from the clock input
    //Periodically toggle LED
    DRV_TMR_Alarm16BitRegister ( tmrHandle, 0xFF00, true,
                                    DebounceCheck );
}

void DebounceCheck ( uintptr_t context )
{
    readKey = AppReadKey();

    if ( readKey != lastReadKey )
    {
        lastReadKey = readKey;
        keyCount = 0;
    }
    else
    {
        if ( keyCount > 20 )
        {
            globalKeyState = readKey;
            DRV_TMR_Alarm16BitDeregister ( tmrHandle );
        }
        keyCount++;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TMR_Alarm16BitDeregister (DRV_HANDLE handle)

DRV_TMR_Alarm32BitDeregister Function

Removes a previously set alarm.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Alarm32BitDeregister(DRV_HANDLE handle);
```

Returns

None.

Description

This function removes a previously set alarm. This API is valid only if the 32-bit mode of the timer is selected Otherwise use DRV_TMR_Alarm16BitDeregister function.

Remarks

In most of the devices only even numbered instances of timer supports 32-bit mode.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 32-Bit timer mode if mode selection is applicable.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

DRV_TMR_Alarm32BitRegister function must have been called before.

Example

```
//Example of a key debounce check

static unsigned int lastReadKey, readKey, keyCount, globalKeyState;
DRV_HANDLE              tmrHandle;  // Returned from DRV_TMR_Open
//Do the initialization with 'mode' set to DRV_TMR_OPERATION_MODE_32_BIT

void keyPressDetect ( void )
{
    // Calculate the count to be passed on from the clock input
    //Periodically check the key status
    DRV_TMR_Alarm32BitRegister ( tmrHandle, 0xFF0FFD20, true, 0,
                                    DebounceCheck );
}

void DebounceCheck ( uintptr_t context )
{
    readKey = AppReadKey();

    if ( readKey != lastReadKey )
    {
        lastReadKey = readKey;
        keyCount = 0;
    }
    else
    {
        if ( keyCount > 20 )
        {
            //Key is stable now
            globalKeyState = readKey;
            DRV_TMR_Alarm32BitDeregister ( tmrHandle );
        }
        keyCount++;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TMR_Alarm32BitDeregister (DRV_HANDLE handle)

DRV_TMR_AlarmHasElapsed Function

Provides the status of Timer's period elapse.

Implementation: Dynamic

File

drv_tmr.h

C

```
unsigned int DRV_TMR_AlarmHasElapsed(DRV_HANDLE handle);
```

Returns

Number of times timer has elapsed since the last call.

Description

This function returns the number of times Timer's period has elapsed since last call to this API has made. On calling this API, the internally maintained counter will be cleared and count will be started again from next elapse.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      tmrHandle;  // Returned from DRV_TMR_Open
bool            elapseStatus;
SYS_MODULE_OBJ  tmrObject // Returned by DRV_TMR_Initialize
unsigned int    appInternalTime = 0;

Sys_Tasks()
{
    //Timer task will be called from ISR

    APP_TimeUpdate_Task();

    //Other Tasks
}

void APP_TimeUpdate_Task ( void )
{
    //We will not miss a count even though we are late
    appInternalTime += DRV_TMR_AlarmHasElapsed ( tmrHandle );
}
```

Parameters

Parameters	Description
handle	A valid handle, returned from the DRV_TMR_Open

Function

unsigned int DRV_TMR_AlarmHasElapsed (DRV_HANDLE handle)

DRV_TMR_AlarmPeriod16BitGet Function

Provides the 16-bit Timer's period.

Implementation: Dynamic

File

drv_tmr.h

C

```
uint16_t DRV_TMR_AlarmPeriod16BitGet(DRV_HANDLE handle);
```

Returns

16-bit timer period value

Description

This function gets the 16-bit Timer's period. This API is valid only if the 16-bit mode of the timer is selected. Otherwise use DRV_TMR_AlarmPeriod32BitGet function.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 16-Bit timer mode if mode selection is applicable.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
uint16_t period;

period = DRV_TMR_AlarmPeriod16BitGet ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint16_t DRV_TMR_AlarmPeriod16BitGet (DRV_HANDLE handle)

DRV_TMR_AlarmPeriod16BitSet Function

Updates the 16-bit Timer's period.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_AlarmPeriod16BitSet(DRV_HANDLE handle, uint16_t value);
```

Returns

None.

Description

This function updates the 16-bit Timer's period. This API is valid only if the 16-bit mode of the timer is selected Otherwise use DRV_TMR_AlarmPeriod32BitSet function.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 16-Bit timer mode if mode selection is applicable.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TMR_Open

DRV_TMR_AlarmPeriod16BitSet ( handle, 0x1000 );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
value	16-bit Period value

Function

void DRV_TMR_AlarmPeriod16BitSet (DRV_HANDLE handle, uint16_t value)

DRV_TMR_AlarmPeriod32BitGet Function

Provides the 32-bit Timer's period.

Implementation: Dynamic

File

drv_tmr.h

C

```
uint32_t DRV_TMR_AlarmPeriod32BitGet(DRV_HANDLE handle);
```

Returns

32-bit Timer period value.

Description

This function gets the 32-bit Timer's period. This API is valid only if the 32-bit mode of the timer is selected Otherwise use DRV_TMR_AlarmPeriod16BitGet function.

Remarks

In most of the devices only even numbered instances of timer supports 32-bit mode.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 32-Bit timer mode.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TMR_Open
uint32_t period;

period = DRV_TMR_AlarmPeriod32BitGet ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_TMR_AlarmPeriod32BitGet (DRV_HANDLE handle)

DRV_TMR_AlarmPeriod32BitSet Function

Updates the 32-bit Timer's period.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_AlarmPeriod32BitSet(DRV_HANDLE handle, uint32_t period);
```

Returns

None.

Description

This function updates the 32-bit Timer's period. This API is valid only if the 32-bit mode of the timer is selected Otherwise use DRV_TMR_AlarmPeriod16BitSet function.

Remarks

In most of the devices only even numbered instances of timer supports 32-bit mode.

Preconditions

The DRV_TMR_Initialize unction must have been called. Must have selected 32-Bit timer mode.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TMR_Open

DRV_TMR_AlarmPeriod32BitSet ( handle, 0xFFFFFF00 );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
period	32-bit Period value

Function

void DRV_TMR_AlarmPeriod32BitSet (DRV_HANDLE handle, uint32_t period)

DRV_TMR_AlarmDisable Function

Disables an alarm signal.

Implementation: Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_AlarmDisable(DRV_HANDLE handle);
```

Returns

The current status of the alarm:

- true if the alarm was currently enabled
- false if the alarm was currently disabled

Description

This function allows the client to disable an alarm generation. Use DRV_TMR_AlarmEnable to re-enable.

Remarks

When the driver operates in interrupts this call resolves to a device interrupt disable.

Do NOT disable the timer except for very short periods of time. If the time that the interrupt is disabled is longer than a wrap around period and the interrupt is missed, the hardware has no means of recovering and the resulting timing will be inaccurate.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

A client alarm must be active.

Example

Parameters

Parameters	Description
handle	A valid handle, returned from DRV_TMR_Open

Function

bool DRV_TMR_AlarmDisable (DRV_HANDLE handle);

DRV_TMR_AlarmEnable Function

Re-enables an alarm signal.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_AlarmEnable(DRV_HANDLE handle, bool enable);
```

Returns

None

Description

This function allows the client to re-enable an alarm after it has been disabled by a DRV_TMR_AlarmDisable call.

Remarks

When the driver operates in interrupts this call resolves to a device interrupt re-enable.

Preconditions

The DRV_TMR_Initialize function must have been called. DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

Parameters

Parameters	Description
handle	A valid handle, returned from DRV_TMR_Open
enable	boolean to enable the current callback

Function

void DRV_TMR_AlarmEnable (DRV_HANDLE handle, bool enable);

DRV_TMR_Alarm16BitRegister Function

Sets up an alarm.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Alarm16BitRegister(DRV_HANDLE handle, uint16_t period, bool isPeriodic, uintptr_t
context, DRV_TMR_CALLBACK callBack);
```

Returns

None

Description

This function sets up an alarm, allowing the client to receive a callback from the driver when the counter period elapses. Alarms can be one-shot or periodic. This API is valid only if the 16-bit mode of the timer is selected. Otherwise use DRV_TMR_Alarm32BitRegister function.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 16-Bit timer mode if mode selection is applicable.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
//Do the initialization with 'mode' set to DRV_TMR_OPERATION_MODE_16_BIT

void setupTask ()
{
    DRV_HANDLE              tmrHandle;  // Returned from DRV_TMR_Open

    // Calculate the count to be passed on from the clock input
    //Periodically toggle LED
    DRV_TMR_Alarm16BitRegister ( tmrHandle, 0xFF40, true, 0,
                                    ToggleLedCallBack );
}

void ToggleLedCallBack ( uintptr_t context )
{
    //Toggle
}
```

Parameters

Parameters	Description
handle	A valid handle, returned from DRV_TMR_Open
period	16-bit period which will be loaded into the Timer hardware register.
isPeriodic	Flag indicating whether the alarm should be one-shot or periodic.
context	A reference, call back function will be called with the same reference.
callBack	A call back function which will be called on period elapse.

Function

void DRV_TMR_Alarm16BitRegister (DRV_HANDLE handle, uint16_t period, bool isPeriodic,
uintptr_t context, DRV_TMR_CALLBACK callBack)

DRV_TMR_Alarm32BitRegister Function

Sets up an alarm.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Alarm32BitRegister(DRV_HANDLE handle, uint32_t period, bool isPeriodic, uintptr_t
context, DRV_TMR_CALLBACK callBack);
```

Returns

None

Description

This function sets up an alarm, allowing the client to receive a callback from the driver when the counter period elapses. Alarms can be one-shot or periodic. This API is valid only if the 32-bit mode of the timer is selected Otherwise use DRV_TMR_Alarm16BitRegister function.

Remarks

In most of the devices only even numbered instances of timer supports 32-bit mode.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 32-Bit timer mode.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
//Do the initialization with 'mode' set to DRV_TMR_OPERATION_MODE_32_BIT

void setupTask ()
{
    DRV_HANDLE              tmrHandle;  // Returned from DRV_TMR_Open

    // Calculate the count to be passed on from the clock input
    //Periodically toggle LED
    DRV_TMR_Alarm32BitRegister ( tmrHandle, 0xFFFFFF00, true, 0,
                                    ToggleLedCallBack );
}

void ToggleLedCallBack ( uintptr_t context )
{
    //Toggle
}
```

Parameters

Parameters	Description
handle	A valid handle, returned from DRV_TMR_Open
period	32-bit period which will be loaded into the Timer hardware register.
isPeriodic	Flag indicating whether the alarm should be one-shot or periodic.
context	A reference, call back function will be called with the same reference.
callBack	A call back function which will be called on period elapse.

Function

void DRV_TMR_Alarm32BitRegister (DRV_HANDLE handle, uint32_t period, bool isPeriodic,
uintptr_t context, DRV_TMR_CALLBACK callBack)

© 2013-2015 Microchip Technology Inc.

DRV_TMR_AlarmDeregister Function

Removes a previously set alarm.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_AlarmDeregister(DRV_HANDLE handle);
```

Returns

None.

Description

This function removes a previously set alarm.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

DRV_TMR_AlarmRegister function must have been called before.

Example

```
// Example of a key debounce check

static unsigned int lastReadKey, readKey, keyCount, globalKeyState;
DRV_HANDLE              tmrHandle;  // Returned from DRV_TMR_Open

void keyPressDetect ()
{
    // Calculate the count to be passed on from the clock input
    DRV_TMR_AlarmRegister ( tmrHandle, 0xFF00, true, DebounceCheck );
}

void DebounceCheck ( uintptr_t context )
{
    readKey = AppReadKey();

    if ( readKey != lastReadKey )
    {
        lastReadKey = readKey;
        keyCount = 0;
    }
    else
    {
        if ( keyCount > 20 )
        {
            globalKeyState = readKey;
            DRV_TMR_AlarmDeregister ( tmrHandle );
        }
        keyCount++;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TMR_AlarmDeregister (DRV_HANDLE handle)

DRV_TMR_AlarmPeriodGet Function

Provides the Timer's period.

Implementation: Dynamic

File

drv_tmr.h

C

```
uint32_t DRV_TMR_AlarmPeriodGet(DRV_HANDLE handle);
```

Returns

Timer period value:

- a 16 bit value if the timer is configured in 16 bit mode
- a 32 bit value if the timer is configured in 32 bit mode

Description

This function gets the Timer's period.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
uint32_t period;

period = DRV_TMR_AlarmPeriodGet ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_TMR_AlarmPeriodGet (DRV_HANDLE handle)

DRV_TMR_AlarmPeriodSet Function

Updates the Timer's period.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_AlarmPeriodSet(DRV_HANDLE handle, uint32_t value);
```

Returns

None.

Description

This function updates the Timer's period.

Remarks

- The period value will be truncated to a 16 bit value if the timer is configured in 16 bit mode.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TMR_Open

DRV_TMR_AlarmPeriodSet ( handle, 0x1000 );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
value	Period value • a 16 bit value if the timer is configured in 16 bit mode • a 32 bit value if the timer is configured in 32 bit mode

Function

void DRV_TMR_AlarmPeriodSet (DRV_HANDLE handle, uint32_t value)

DRV_TMR_AlarmRegister Function

Sets up an alarm.

Implementation: Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_AlarmRegister(DRV_HANDLE handle, uint32_t divider, bool isPeriodic, uintptr_t
context, DRV_TMR_CALLBACK callBack);
```

Returns

- true - if the call succeeded
- false - the obtained divider could not be obtained or the passed handle was invalid

Description

This function sets up an alarm, allowing the client to receive a callback from the driver when the timer counter reaches zero. Alarms can be one-shot or periodic. A periodic alarm will reload the timer and generate alarm until stopped. The alarm frequency is: DRV_TMR_CounterFrequencyGet() / divider;

Remarks

- The divider value will be truncated to a 16 bit value if the timer is configured in 16 bit mode.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

divider value has to be within the timer divider range (see DRV_TMR_DividerSpecGet).

Example

```
//Do the initialization with 'mode' set to DRV_TMR_OPERATION_MODE_16_BIT

void setupTask ()
{
    DRV_HANDLE              tmrHandle;  // Returned from DRV_TMR_Open

    uint32_t myFreq = 1000; // 1KHz
    uint32_t clkFreq = DRV_TMR_CounterFrequencyGet(tmrHandle);   // timer running frequency

    // calculate the divider needed
    uint32_t divider = clkFreq / myFreq;

    // Start the alarm
    if(!DRV_TMR_AlarmRegister ( tmrHandle, divider, true, 0, CallBackFreq ))
    {
        // divider value could not be obtain;
        // handle the error
        //
    }
}
```

Parameters

Parameters	Description
handle	A valid handle, returned from DRV_TMR_Open
divider	The value to divide the timer clock source to obtain the required alarm frequency. • a 16 bit value if the timer is configured in 16 bit mode • a 32 bit value if the timer is configured in 32 bit mode

isPeriodic	Flag indicating whether the alarm should be one-shot or periodic.
context	A reference, call back function will be called with the same reference.
callBack	A call back function which will be called on time out.

Function

```
bool DRV_TMR_AlarmRegister ( DRV_HANDLE handle, uint32_t divider, bool isPeriodic,
uintptr_t context, DRV_TMR_CALLBACK callBack )
```

d) Counter Control Functions

DRV_TMR_CounterFrequencyGet Function

Provides the Timer input frequency.

Implementation: Dynamic

File

drv_tmr.h

C

```
uint32_t DRV_TMR_CounterFrequencyGet(DRV_HANDLE handle);
```

Returns

32-bit value corresponding to the running frequency.

Description

This function provides the Timer input frequency. Input frequency is the clock to the Timer register and it is considering the prescaler divisor.

Remarks

On most processors, the Timer's base frequency is the same as the peripheral bus clock.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
uint32_t   clkFreqHz;

clkFreqHz = DRV_TMR_CounterFrequencyGet ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_TMR_CounterFrequencyGet (DRV_HANDLE handle)

DRV_TMR_CounterValue16BitGet Function

Reads the 16-bit Timer's counter register.

Implementation: Dynamic

File

drv_tmr.h

C

```
uint16_t DRV_TMR_CounterValue16BitGet(DRV_HANDLE handle);
```

Returns

Timer period in 16-bit mode.

Description

This function returns the 16-bit Timer's value in the counter register. This is valid only if the 16-bit mode of the timer is selected. Otherwise use DRV_TMR_CounterValue32BitGet function.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 16-Bit timer mode if mode selection is applicable.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
//Example to use timer for precision time measurement
//without configuring an alarm (interrupt based)
char appState = 0;
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

switch ( appState )
{
    case 0:
        //Calculate and set the counter period
        DRV_TMR_CounterValue16BitSet ( tmrHandle, ( 0xFFFF - 0x1000 ) );

        //counter starts
        DRV_TMR_Start ( tmrHandle );

        //Trigger an application operation
        app_trigger_operation();

        //Check for time-out in the next state
        appState++;
    case 1:
        //Overflows and stops at 0 if no alarm is set
        if ( DRV_TMR_CounterValue16BitGet ( tmrHandle ) == 0 )
        {
            //Time-out
            return false;
        }
        else if ( app_operation_isComplete( ) )
        {
            //Operation is complete before time-out
            return true;
        }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint16_t DRV_TMR_CounterValue16BitGet (DRV_HANDLE handle)

DRV_TMR_CounterValue16BitSet Function

Updates the 16-bit Timer's counter register.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_CounterValue16BitSet(DRV_HANDLE handle, uint16_t counterPeriod);
```

Returns

None.

Description

This function updates the 16-bit Timer's value in the counter register. This is valid only if the 16-bit mode of the timer is selected('mode' in the INIT structure is set to DRV_TMR_OPERATION_MODE_16_BIT). Otherwise use DRV_TMR_CounterValue32BitSet function.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 16-Bit timer mode if mode selection is applicable.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
//Example to use timer for precision time measurement
//without configuring an alarm (interrupt based)
char appState = 0;
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

switch ( appState )
{
    case 0:
        //Calculate and set the counter period
        DRV_TMR_CounterValue16BitSet ( tmrHandle, ( 0xFFFF - 0x1000 ) );

        //counter starts
        DRV_TMR_Start ( tmrHandle );

        //Trigger an application operation
        app_trigger_operation();

        //Check for time-out in the next state
        appState++;
    case 1:
        //Overflows and stops at 0 if no alarm is set
        if ( DRV_TMR_CounterValue16BitGet ( tmrHandle ) == 0 )
        {
            //Time-out
            return false;
        }
        else if ( app_operation_isComplete( ) )
        {
            //Operation is complete before time-out
            return true;
        }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
counterPeriod	16-bit counter period value

Function

void DRV_TMR_CounterValue16BitSet (DRV_HANDLE handle, uint16_t counterPeriod)

DRV_TMR_CounterValue32BitGet Function

Reads the 32-bit Timer's counter register.

Implementation: Dynamic

File

drv_tmr.h

C

```
uint32_t DRV_TMR_CounterValue32BitGet(DRV_HANDLE handle);
```

Returns

32-Bit Counter value.

Description

This function returns the 32-bit Timer's value in the counter register. This is valid only if the 32-bit mode of the timer is selected Otherwise use DRV_TMR_CounterValue16BitGet function.

Remarks

In most of the devices only even numbered instances of timer supports 32-bit mode.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 32-Bit timer mode.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
//Example to use timer for precision time measurement
//without configuring an alarm (interrupt based)
char appState = 0;
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

switch ( appState )
{
    case 0:
        //Calculate and set the counter period
        DRV_TMR_CounterValue32BitSet ( tmrHandle, ( 0xFFFFFFFF - 0x23321000 ) );

        //counter starts
        DRV_TMR_Start ( tmrHandle );

        //Trigger an application operation
        app_trigger_operation();

        //Check for time-out in the next state
        appState++;
    case 1:
        //Overflows and stops at 0 if no alarm is set
        if ( DRV_TMR_CounterValue32BitGet ( tmrHandle ) == 0 )
        {
            //Time-out
            return false;
        }
        else if ( app_operation_isComplete( ) )
        {
            //Operation is complete before time-out
            return true;
        }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_TMR_CounterValue32BitGet (DRV_HANDLE handle)

DRV_TMR_CounterValue32BitSet Function

Updates the 32-bit Timer's counter register.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_CounterValue32BitSet(DRV_HANDLE handle, uint32_t counterPeriod);
```

Returns

None.

Description

This function updates the 32-bit Timer's value in the counter register. This is valid only if the 32-bit mode of the timer is selected Otherwise use DRV_TMR_CounterValue32BitSet function.

Remarks

In most of the devices only even numbered instances of timer supports 32-bit mode.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 32-Bit timer mode.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
//Example to use timer for precision time measurement
//without configuring an alarm (interrupt based)
char appState = 0;
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

switch ( appState )
{
    case 0:
        //Calculate and set the counter period
        DRV_TMR_CounterValue32BitSet ( tmrHandle, ( 0xFFFFFFFF - 0xFF343100 ) );

        //counter starts
        DRV_TMR_Start ( tmrHandle );

        //Trigger an application operation against which we want to use the time-out
        app_trigger_operation();

        //Check for time-out in the next state
        appState++;
    case 1:
        //Overflows and stops at 0 if no alarm is set
        if ( DRV_TMR_CounterValue32BitGet ( tmrHandle ) == 0 )
        {
            //Time-out
            return false;
        }
        else if ( app_operation_isComplete( ) )
        {
            //Operation is complete before time-out
            return true;
        }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

counterPeriod	32-bit counter period value

Function

void DRV_TMR_CounterValue32BitSet (DRV_HANDLE handle, uint32_t counterPeriod)

DRV_TMR_CounterClear Function

Clears the Timer's counter register.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_CounterClear(DRV_HANDLE handle);
```

Returns

None.

Description

This function clears the Timer's value in the counter register.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TMR_CounterClear (DRV_HANDLE handle)

DRV_TMR_CounterValueGet Function

Reads the Timer's counter register.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
uint32_t DRV_TMR_CounterValueGet(DRV_HANDLE handle);
```

Returns

Timer current period:

- a 16 bit value if the timer is configured in 16 bit mode
- a 32 bit value if the timer is configured in 32 bit mode

Description

This function returns the Timer's value in the counter register.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
//Example to use timer for precision time measurement
//without configuring an alarm (interrupt based)
char appState = 0;
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

switch ( appState )
{
    case 0:
        //Calculate and set the counter period
        DRV_TMR_CounterValueSet ( tmrHandle, ( 0xFFFF - 0x1000 ) );

        //counter starts
        DRV_TMR_Start ( tmrHandle );

        //Trigger an application operation
        app_trigger_operation();

        //Check for time-out in the next state
        appState++;
    case 1:
        //Overflows and stops at 0 if no alarm is set
        if ( DRV_TMR_CounterValueGet ( tmrHandle ) == 0 )
        {
            //Time-out
            return false;
        }
        else if ( app_operation_isComplete( ) )
        {
            //Operation is complete before time-out
            return true;
        }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_TMR_CounterValueGet (DRV_HANDLE handle)

DRV_TMR_CounterValueSet Function

Updates the Timer's counter register.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_CounterValueSet(DRV_HANDLE handle, uint32_t counterPeriod);
```

Returns

None.

Description

This function updates the Timer's value in the counter register.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
counterPeriod	counter period value • a 16 bit value if the timer is configured in 16 bit mode • a 32 bit value if the timer is configured in 32 bit mode

Function

void DRV_TMR_CounterValueSet (DRV_HANDLE handle, uint32_t counterPeriod)

e) Miscellaneous Functions

DRV_TMR_GateModeClear Function

Enables the Gate mode.

Implementation: Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_GateModeClear(DRV_HANDLE handle);
```

Returns

- true - if the operation is successful
- false - either the handle is invalid or the gate mode is not supported

Description

This function enables the Gated mode of Timer. User can measure the duration of an external signal in this mode. Once the Gate mode is enabled, Timer will start on the raising edge of the external signal. It will keep counting until the next falling edge.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

DRV_TMR_GateModeClear ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

bool DRV_TMR_GateModeClear (DRV_HANDLE handle)

DRV_TMR_PrescalerGet Function

This function gets the currently selected prescaler.

Implementation: Dynamic

File

drv_tmr.h

C

```
TMR_PRESCALE DRV_TMR_PrescalerGet(DRV_HANDLE handle);
```

Returns

Timer prescaler.

Description

This function gets the currently selected prescaler.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
TMR_PRESCALE preScale;

preScale = DRV_TMR_PrescalerGet ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

TMR_PRESCALE DRV_TMR_PrescalerGet (DRV_HANDLE handle)

DRV_TMR_OperationModeGet Function

This function gets the currently selected operation mode.

Implementation: Dynamic

File

drv_tmr.h

C

```
DRV_TMR_OPERATION_MODE DRV_TMR_OperationModeGet(DRV_HANDLE handle);
```

Returns

A DRV_TMR_OPERATION_MODE value showing how the timer is currently configured.
DRV_TMR_OPERATION_MODE_NONE is returned for an invalid client handle.

Description

This function gets the currently selected 16/32 bit operation mode.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
DRV_TMR_OPERATION_MODE operMode;

operMode = DRV_TMR_OperationModeGet ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_TMR_OPERATION_MODE DRV_TMR_OperationModeGet(DRV_HANDLE handle)

DRV_TMR_DividerRangeGet Function

Returns the Timer divider values.

Implementation: Dynamic

File

drv_tmr.h

C

```
DRV_TMR_OPERATION_MODE DRV_TMR_DividerRangeGet(DRV_HANDLE handle, DRV_TMR_DIVIDER_RANGE*
pDivRange);
```

Returns

- A DRV_TMR_OPERATION_MODE value showing how the timer is currently configured. The pDivRange is updated with the supported range values.

- DRV_TMR_OPERATION_MODE_NONE for invalid client handle

Description

This function provides the Timer operating mode and divider range.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
DRV_TMR_OPERATION_MODE  timerMode;
DRV_TMR_DIVIDER_RANGE   timerRange;

DRV_TMR_DividerRangeGet(handle, &timerRange);
uint32_t   clkFreqHz = DRV_TMR_CounterFrequencyGet ( tmrHandle );

uint32_t    maxFreqHz = clkFreqHz / timerRange.dividerMin;
uint32_t    minFreqHz = clkFreqHz / timerRange.dividerMax;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
pDivRange	Address to store the timer divider range.

Function

DRV_TMR_OPERATION_MODE DRV_TMR_DividerRangeGet (DRV_HANDLE handle,
DRV_TMR_DIVIDER_RANGE* pDivRange)

DRV_TMR_Tasks_ISR Function

Maintains the driver's state machine, processes the events and implements its ISR.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Tasks_ISR(SYS_MODULE_OBJ object);
```

Returns

None

Description

This function is used to maintain the driver's internal state machine and processes the timer events in interrupt-driven implementations (DRV_TMR_INTERRUPT_MODE == true).

Remarks

This function is normally not called directly by an application. It is called by the timer driver raw ISR.

This function will execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_TMR_Initialize function must have been called for the specified Timer driver instance.

Example

```
void __ISR(_TIMER_2_VECTOR, ipl4) _InterruptHandler_TMR2(void)
{
    DRV_TMR_Tasks_ISR(appDrvObjects.drvTmrObject);
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_TMR_Initialize)

Function

void DRV_TMR_Tasks_ISR (SYS_MODULE_OBJ object)

f) Data Types and Constants

DRV_TMR_CALLBACK Type

Pointer to a Timer driver callback function data type.

File

drv_tmr.h

C

```
typedef void (* DRV_TMR_CALLBACK)(uintptr_t context, uint32_t alarmCount);
```

Description

Timer Driver Callback Function Pointer

This data type defines a pointer to a Timer driver callback function.

Remarks

Useful only when timer alarm callback support is enabled by defining the DRV_TMR_ALARM_ENABLE configuration option.

DRV_TMR_INIT Structure

Defines the Timer driver initialization data.

File

drv_tmr.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  TMR_MODULE_ID tmrId;
  DRV_TMR_CLK_SOURCES clockSource;
  TMR_PRESCALE prescale;
  INT_SOURCE interruptSource;
  DRV_TMR_OPERATION_MODE mode;
  bool asyncWriteEnable;
} DRV_TMR_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization.
TMR_MODULE_ID tmrId;	Identifies timer hardware module (PLIB-level) ID
DRV_TMR_CLK_SOURCES clockSource;	Clock Source select.
TMR_PRESCALE prescale;	Prescaler Selection from the processor enumeration
INT_SOURCE interruptSource;	Interrupt Source for TMR module. If 'DRV_TMR_OPERATION_MODE_32_BIT' flag is selected the interrupt will be generated by the 2nd timer of the pair, the odd numbered one.
DRV_TMR_OPERATION_MODE mode;	Select 16/32 bit operation mode. 32 bit mode will combine two 16 bit timer modules to form a 32 bit one. This is usually only necessary for very long delays.
bool asyncWriteEnable;	Asynchronous write enable configuration. If true the asynchronous write is enabled. For timers that do not support this feature the value is ignored

Description

Timer Driver Initialize Data

This data type defines data required to initialize the Timer driver.

Remarks

Not all initialization features are available on all devices.

DRV_TMR_CLIENT_STATUS Enumeration

Identifies the client-specific status of the Timer driver

File

drv_tmr.h

C

```
typedef enum {
  DRV_TMR_CLIENT_STATUS_INVALID,
  DRV_TMR_CLIENT_STATUS_BUSY,
  DRV_TMR_CLIENT_STATUS_READY,
  DRV_TMR_CLIENT_STATUS_RUNNING
} DRV_TMR_CLIENT_STATUS;
```

Members

Members	Description
DRV_TMR_CLIENT_STATUS_INVALID	Driver is invalid (or unopened) state
DRV_TMR_CLIENT_STATUS_BUSY	An operation is currently in progress
DRV_TMR_CLIENT_STATUS_READY	Ready, no operations running
DRV_TMR_CLIENT_STATUS_RUNNING	Timer started and running, processing transactions

Description

Timer Driver Client Status

This enumeration identifies the client-specific status of the Timer driver.

Remarks

None.

DRV_TMR_CLK_SOURCES Enumeration

Lists the clock sources available for timer driver.

File

drv_tmr.h

C

```
typedef enum {
  DRV_TMR_CLKSOURCE_INTERNAL,
  DRV_TMR_CLKSOURCE_EXTERNAL_SYNCHRONOUS,
  DRV_TMR_CLKSOURCE_EXTERNAL_ASYNCHRONOUS
} DRV_TMR_CLK_SOURCES;
```

Members

Members	Description
DRV_TMR_CLKSOURCE_INTERNAL	Clock input to the timer module is internal(Peripheral Clock)
DRV_TMR_CLKSOURCE_EXTERNAL_SYNCHRONOUS	Clock input to the timer module is external with clock clock synchronization enabled
DRV_TMR_CLKSOURCE_EXTERNAL_ASYNCHRONOUS	Clock input to the timer module is external with clock clock synchronization disabled

Description

Timer Driver Clock sources

This enumeration lists all the available clock sources for the timer hardware.

Remarks

Not all modes are available on all devices.

'Synchronization' may not be applicable for all the instances of the timer. The driver discards the Synchronization mode selected if it is not applicable for the selected hardware.

DRV_TMR_DIVIDER_RANGE Structure

This data structure specifies the divider values that can be obtained by the timer module.

File

drv_tmr.h

C

```
typedef struct {
  uint32_t dividerMin;
  uint32_t dividerMax;
  uint32_t dividerStep;
} DRV_TMR_DIVIDER_RANGE;
```

Members

Members	Description
uint32_t dividerMin;	The minimum divider value that the timer module can obtain
uint32_t dividerMax;	The maximum divider value that the timer module can obtain
uint32_t dividerStep;	The divider step value, between 2 divider values Should be 1 for most timers

Description

Timer Driver divider operating specification

This data structure specifies the divider values that can be obtained by the timer hardware.

Remarks

None.

DRV_TMR_OPERATION_MODE Enumeration

Lists the operation modes available for timer driver.

File

drv_tmr.h

C

```
typedef enum {
  DRV_TMR_OPERATION_MODE_NONE,
  DRV_TMR_OPERATION_MODE_16_BIT,
  DRV_TMR_OPERATION_MODE_32_BIT
} DRV_TMR_OPERATION_MODE;
```

Members

Members	Description
DRV_TMR_OPERATION_MODE_NONE	The timer module operating mode none/invalid
DRV_TMR_OPERATION_MODE_16_BIT	The timer module operates in 16 bit mode
DRV_TMR_OPERATION_MODE_32_BIT	The timer module operates in 32 bit mode This will combine two 16 bit timer modules

Description

Timer Driver Operation mode

This enumeration lists all the available operation modes that are valid for the timer hardware.

Remarks

Not all modes are available on all devices.

DRV_TMR_INDEX_COUNT Macro

Number of valid Timer driver indices.

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_COUNT TMR_NUMBER_OF_MODULES
```

Description

Timer Driver Module Index Count

This constant identifies Timer driver index definitions.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is device-specific.

DRV_TMR_INDEX_0 Macro

Timer driver index definitions

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_0 0
```

Description

Timer Driver Module Index Numbers

These constants provide Timer driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_TMR_Initialize and DRV_TMR_Open functions to identify the driver instance in use.

DRV_TMR_INDEX_1 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_1 1
```

Description

This is macro DRV_TMR_INDEX_1.

DRV_TMR_INDEX_2 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_2 2
```

Description

This is macro DRV_TMR_INDEX_2.

© 2013-2015 Microchip Technology Inc.

DRV_TMR_INDEX_3 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_3 3
```

Description

This is macro DRV_TMR_INDEX_3.

DRV_TMR_INDEX_4 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_4 4
```

Description

This is macro DRV_TMR_INDEX_4.

DRV_TMR_INDEX_5 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_5 5
```

Description

This is macro DRV_TMR_INDEX_5.

DRV_TMR_INDEX_6 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_6 6
```

Description

This is macro DRV_TMR_INDEX_6.

DRV_TMR_INDEX_7 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_7 7
```

Description

This is macro DRV_TMR_INDEX_7.

© 2013-2015 Microchip Technology Inc.

DRV_TMR_INDEX_8 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_8 8
```

Description

This is macro DRV_TMR_INDEX_8.

DRV_TMR_INDEX_9 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_9 9
```

Description

This is macro DRV_TMR_INDEX_9.

Files

Files

Name	Description
drv_tmr.h	Timer device driver interface header file.
drv_tmr_config_template.h	Timer driver configuration definitions for the template version.

Description

This section lists the source and header files used by the Timer Driver Library.

drv_tmr.h

Timer device driver interface header file.

Enumerations

	Name	Description
	DRV_TMR_CLIENT_STATUS	Identifies the client-specific status of the Timer driver
	DRV_TMR_CLK_SOURCES	Lists the clock sources available for timer driver.
	DRV_TMR_OPERATION_MODE	Lists the operation modes available for timer driver.

Functions

	Name	Description
	DRV_TMR_Alarm16BitDeregister	Removes a previously set alarm. **Implementation:** Dynamic
	DRV_TMR_Alarm16BitRegister	Sets up an alarm. **Implementation:** Dynamic
	DRV_TMR_Alarm32BitDeregister	Removes a previously set alarm. **Implementation:** Dynamic
	DRV_TMR_Alarm32BitRegister	Sets up an alarm. **Implementation:** Dynamic
	DRV_TMR_AlarmDeregister	Removes a previously set alarm. **Implementation:** Dynamic
	DRV_TMR_AlarmDisable	Disables an alarm signal. **Implementation:** Dynamic
	DRV_TMR_AlarmEnable	Re-enables an alarm signal. **Implementation:** Dynamic
	DRV_TMR_AlarmHasElapsed	Provides the status of Timer's period elapse. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriod16BitGet	Provides the 16-bit Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriod16BitSet	Updates the 16-bit Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriod32BitGet	Provides the 32-bit Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriod32BitSet	Updates the 32-bit Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriodGet	Provides the Timer's period. **Implementation:** Dynamic

	Name	Description
	DRV_TMR_AlarmPeriodSet	Updates the Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmRegister	Sets up an alarm. **Implementation:** Dynamic
	DRV_TMR_ClientStatus	Gets the status of the client operation. **Implementation:** Dynamic
	DRV_TMR_ClockSet	Sets the timers clock by selecting the source and prescaler. **Implementation:** Dynamic
	DRV_TMR_Close	Closes an opened instance of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_CounterClear	Clears the Timer's counter register. **Implementation:** Static/Dynamic
	DRV_TMR_CounterFrequencyGet	Provides the Timer input frequency. **Implementation:** Dynamic
	DRV_TMR_CounterValue16BitGet	Reads the 16-bit Timer's counter register. **Implementation:** Dynamic
	DRV_TMR_CounterValue16BitSet	Updates the 16-bit Timer's counter register. **Implementation:** Dynamic
	DRV_TMR_CounterValue32BitGet	Reads the 32-bit Timer's counter register. **Implementation:** Dynamic
	DRV_TMR_CounterValue32BitSet	Updates the 32-bit Timer's counter register. **Implementation:** Dynamic
	DRV_TMR_CounterValueGet	Reads the Timer's counter register. **Implementation:** Static/Dynamic
	DRV_TMR_CounterValueSet	Updates the Timer's counter register. **Implementation:** Static/Dynamic
	DRV_TMR_Deinitialize	Deinitializes the specified instance of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_DividerRangeGet	Returns the Timer divider values. **Implementation:** Dynamic
	DRV_TMR_GateModeClear	Enables the Gate mode. **Implementation:** Dynamic
	DRV_TMR_GateModeSet	Enables the Gate mode. **Implementation:** Dynamic
	DRV_TMR_Initialize	Initializes the Timer driver. **Implementation:** Static/Dynamic
	DRV_TMR_Open	Opens the specified Timer driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_TMR_OperationModeGet	This function gets the currently selected operation mode. **Implementation:** Dynamic
	DRV_TMR_PrescalerGet	This function gets the currently selected prescaler. **Implementation:** Dynamic
	DRV_TMR_Start	Starts the Timer counting. **Implementation:** Static/Dynamic
	DRV_TMR_Status	Provides the current status of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_Stop	Stops the Timer from counting. **Implementation:** Static/Dynamic

	DRV_TMR_Tasks	Maintains the driver's state machine. **Implementation:** Dynamic
	DRV_TMR_Tasks_ISR	Maintains the driver's state machine, processes the events and implements its ISR. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_TMR_INDEX_0	Timer driver index definitions
	DRV_TMR_INDEX_1	This is macro DRV_TMR_INDEX_1.
	DRV_TMR_INDEX_2	This is macro DRV_TMR_INDEX_2.
	DRV_TMR_INDEX_3	This is macro DRV_TMR_INDEX_3.
	DRV_TMR_INDEX_4	This is macro DRV_TMR_INDEX_4.
	DRV_TMR_INDEX_5	This is macro DRV_TMR_INDEX_5.
	DRV_TMR_INDEX_6	This is macro DRV_TMR_INDEX_6.
	DRV_TMR_INDEX_7	This is macro DRV_TMR_INDEX_7.
	DRV_TMR_INDEX_8	This is macro DRV_TMR_INDEX_8.
	DRV_TMR_INDEX_9	This is macro DRV_TMR_INDEX_9.
	DRV_TMR_INDEX_COUNT	Number of valid Timer driver indices.

Structures

	Name	Description
	DRV_TMR_DIVIDER_RANGE	This data structure specifies the divider values that can be obtained by the timer module.
	DRV_TMR_INIT	Defines the Timer driver initialization data.

Types

	Name	Description
	DRV_TMR_CALLBACK	Pointer to a Timer driver callback function data type.

Description

Timer Device Driver Interface Definition

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the Timer device driver.

File Name

drv_tmr.h

Company

Microchip Technology Inc.

drv_tmr_config_template.h

Timer driver configuration definitions for the template version.

Macros

	Name	Description
	DRV_TMR_ASYNC_WRITE_ENABLE	Controls Asynchronous Write mode of the Timer.
	DRV_TMR_CLIENTS_NUMBER	Sets up the maximum number of clients that can be supported by an instance of the dynamic driver.

	DRV_TMR_CLOCK_PRESCALER	Sets the default timer driver clock prescaler.
	DRV_TMR_CLOCK_SOURCE	Sets the default timer driver clock source.
	DRV_TMR_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_TMR_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_TMR_INTERRUPT_SOURCE	Sets the default timer driver clock interrupt source
	DRV_TMR_MODE	Sets the default timer driver clock operating mode.
	DRV_TMR_MODULE_ID	Sets the default timer module ID to be used by the timer driver.
	DRV_TMR_MODULE_INIT	Sets the default module init value for the timer driver.

Description

Timer Driver Configuration Definitions for the Template Version

These definitions set up the driver for the default mode of operation of the driver.

File Name

drv_tmr_config_template.h

Company

Microchip Technology Inc.

USART Driver Library

This topic describes the USART Driver Library.

Introduction

This section introduces the MPLAB Harmony USART Driver.

Description

The MPLAB Harmony USART Driver (also referred to as the USART Driver) provides a high-level interface to the USART and UART peripherals on Microchip's PIC microcontrollers. This driver provides application ready routines to read and write data to the UART using common data transfer models, thus minimizing application overhead. The USART driver features the following:

- Provides byte by byte, read/write and buffer queue data transfer models
- Supports interrupt and Polled modes of operation
- Supports point to point and addressed type data communication
- Support multi-client and multi-instance operation.
- Provides data transfer events
- Supports blocking and non-blocking operation
- Features thread safe functions for use in RTOS applications
- Supports DMA transfers
- Supports high baud rate setting
- Major features are implemented in separate source code files and can be included only if needed. This helps optimize overall application code size.

Using the Library

This topic describes the basic architecture of the USART Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_usart.h`

The interface to the USART library is defined in the `drv_usart.h` header file.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the USART Driver Library.

Library Interface Section	Description
System Routines	These routines are accessed by the MPLAB Harmony System Module. They allow the driver to be initialized, deinitialized and maintained.
Core Client Routines	These routines allow the application client to Open and Close the driver.
Communication Management Client Routines	These routines allow the client to change the properties of the communication channel (such as baud, parity, etc.).
Buffer Queue Read Write Client Routines	These routines allow the client to use the Buffer Queue Data Transfer Model.
File I/O type Read Write Routines	These routines allow the client to use the File I/O type Read Write Routines.
Byte Transfer Routines	These routines allow the client to use the Byte Data Transfer Model.

The USART driver must be first initialized. One or more application clients can then open the USART Driver in blocking or non-blocking mode. The Open function returns a handle which allows the client to access the driver client functionality. The Driver tasks routines should be invoked regularly from the SYS_Tasks routine in case of Polled mode operation or from USART Driver Interrupt Service Routine, in case of Interrupt mode.

The driver implementation is split across multiple files to optimize the application project code size. The application project must include the `drv_usart.c` file if the USART driver is needed in the application. If DMA-enabled data transfers are required, the `drv_usart_dma.c` file should be included into the project instead of the `drv_usart.c` file. These files implement the System and Core Client routines. Other driver files can be included based on the required driver features.

The USART Driver API, unless otherwise specified, should not be called from an interrupt context. That is, they should not be called from an Interrupt Service Routine (ISR) or they should not be called from event handlers that are executing within an ISR context.

Abstraction Model

This section describes how the USART Driver abstracts the USART peripheral features.

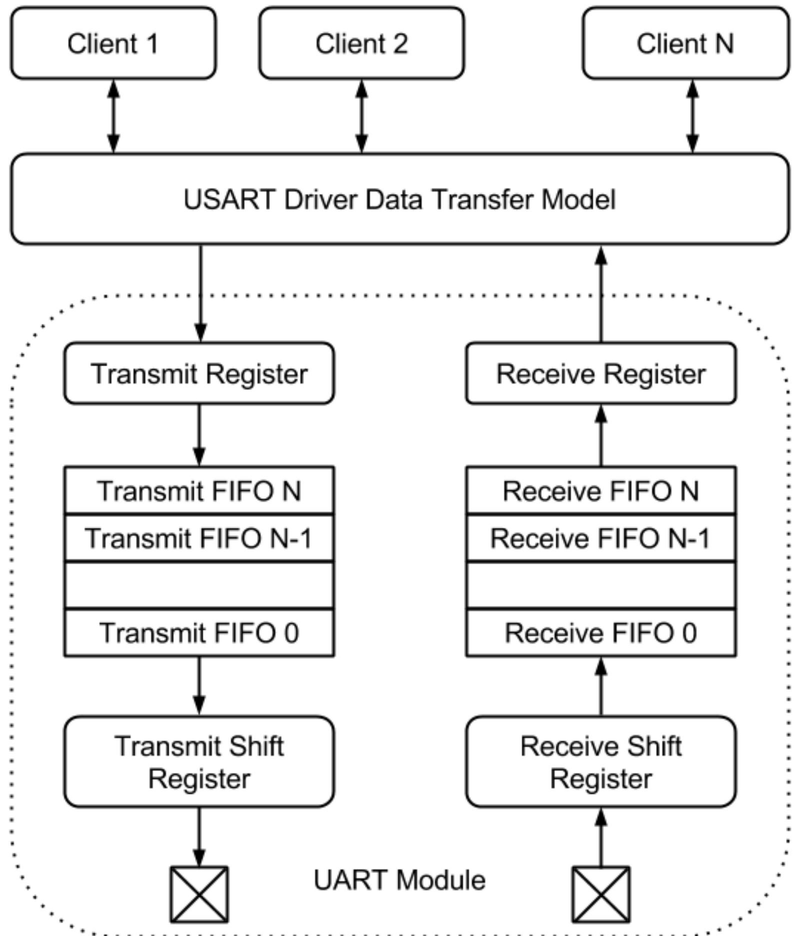
Description

The USART driver features routines to perform the following functions:

- Driver initialization
- Transfer data

- Manage communication properties of the module

The Driver initialization routines allow the application to initialize the driver. The driver must be initialized before it can be used by the application. The data transfer routines allow the application to receive and transmit data through the USART. The driver also provides routines to change the communication properties such as USART Baud or Line Control settings.



As seen in the previous figure, the USART driver clients transfer data through the USART Driver Data Transfer model. The driver abstracts out the hardware details of the UART module FIFO mechanism and shift registers, and provides a low overhead data transfer mechanism to the application. The USART driver provides three different data transfer models for transferring data.

- The Byte by Byte Model
- The File I/O Type Read/Write Transfer Model
- Buffer Queue Transfer Model

Byte by Byte Model:

The Byte by Byte Model allows the application to transfer data through USART driver one byte at a time. With this model, the driver reads one byte from the receive FIFO or writes one byte to the transmit FIFO. The application must check if data has been received before reading the data. Similarly, it must check if the transmit FIFO is not full before writing to the FIFO. The byte by byte data transfer model places the responsibility of maintaining the USART peripheral on the Application. The driver cannot support other data transfer models if support for this data transfer model is enabled. The Byte by Byte data transfer model can be used for simple data transfer applications.

To use the Byte-by-Byte Data Transfer model, the `drv_usart_byte_model.c` file must be included in the project and the DRV_USART_BYTE_MODEL_SUPPORT configuration macro should be set to true.

File I/O Type Read/Write Transfer Model:

This data transfer model is similar to file read and write model in an UNIX operating system. The application calls the USART driver read and write routines to transfer data through the USART. Unlike the Byte by Byte data transfer model, the Read/Write data model can process a block of data. Depending on the mode (blocking or non-blocking) in which the client opened the driver, the driver will either block until all of the data is transferred or will immediately return with the number of bytes transferred. The application does not have to check the FIFO status while using this mode. The application can instead use the return status (number of bytes transferred) to maintain its logic and throttle the data transfer to the USART driver. The Read/Write model can be used with the non-DMA Buffer Queue model. It cannot be used with the Byte-by-Byte Model and the DMA-enabled Buffer Queue Model in the same application.

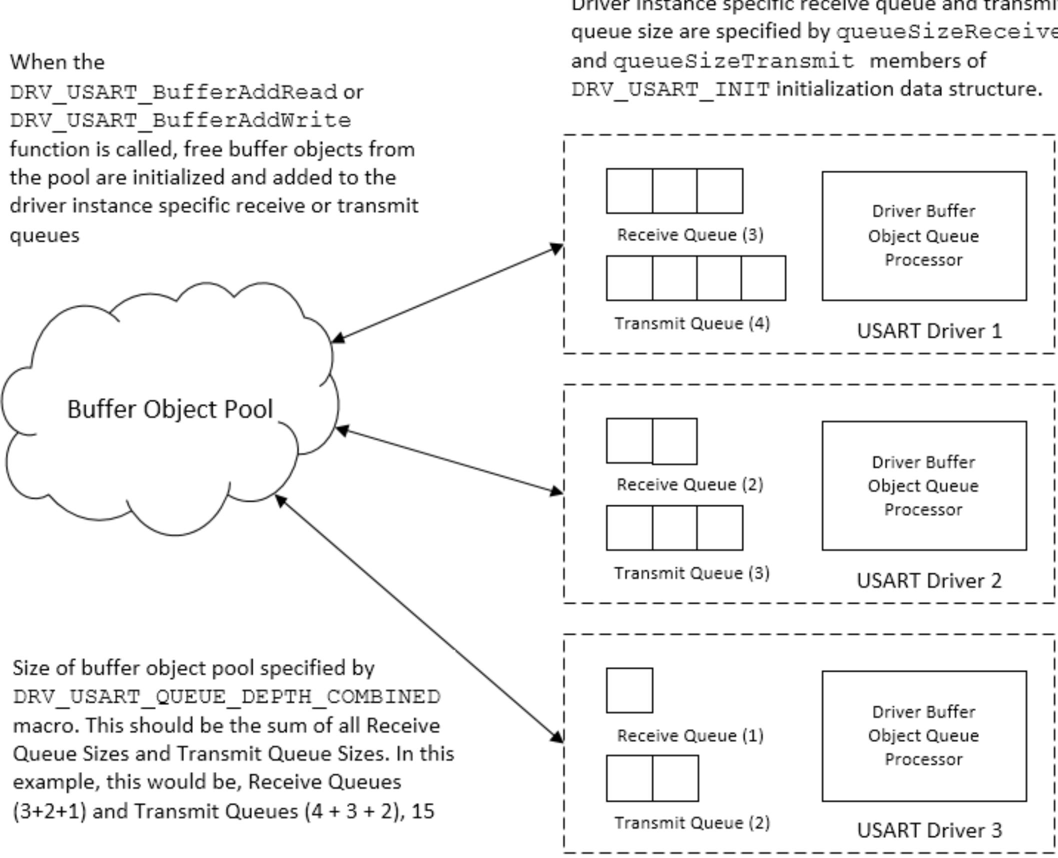
To use the File I/O Type Read/Write Data Transfer model, the `drv_usart_read_write.c` file must be included in the project and the DRV_USART_READ_WRITE_MODEL_SUPPORT configuration macro should be set to true.

See File I/O Type Read Write/Data Transfer Model for additional information.

Buffer Queue Transfer Model:

The Buffer Queue Data Transfer Model allows clients to queue data transfers for processing. This data transfer model is always non-blocking. The USART driver returns a buffer handle for a queued request. The clients can track the completion of a buffer through events and API. If the USART driver is busy processing a data transfer, other data transfer requests are queued. This allows the clients to optimize their application logic and increase throughput. To optimize memory usage, the USART driver implements a shared buffer object pool concept to add a data transfer request to the queue. The following figure shows a conceptual representation of the Buffer Queue Model.

Buffer Queue Transfer Model



As shown in the previous figure, each USART driver hardware instance has a read and write queue. The application must configure the sizes of these read and write queues. The USART driver additionally employs a global pool of buffer queue objects. This pool is common to all USART Driver hardware instances and its size is defined by the DRV_USART_QUEUE_DEPTH_COMBINED configuration macro. When a client places a request to add a data

transfer, the driver performs the following actions:

- It checks if a buffer object is free in the global pool. If not, the driver rejects the request.
- It then checks if the hardware instance specific queue is full. If not, the buffer object from the global pool is added to the hardware instance specific queue. If the queue is full, the driver rejects the request.

The buffer queue model can be used along with the File I/O type Read/Write Data Transfer Model.

To use the Buffer Queue Data Transfer model, the `drv_usart_buffer_queue.c` file must be included in the project and DRV_USART_BUFFER_QUEUE_SUPPORT configuration macro should be set to true.

The USART Driver DMA feature is only available while using the Buffer Queue Model. If enabled, the USART Driver uses the DMA module channels to transfer data directly from application memory to USART transmit or receive registers. This reduces CPU resource consumption and improves system performance.

See Buffer Queue Transfer Model for additional information.

Communication Management

The USART Driver API contains function to control the USART Driver communication properties. These functions allow the client to change the Parity, Stop bits, number of Data bits and the Communication Baud. A change in the communication setting affects all ongoing communication and all driver clients. The `drv_usart_line_control.c` file must be included in the project to use these functions.

How the Library Works

This section describes how to use the USART Driver.

Description

Prior to using the USART driver, the application must decide on which USART data transfer models are required. The application project should then include the USART driver files, required to support the data transfer model into the application project. Additionally, the application design must consider the need for USART driver to be opened in blocking or non blocking modes. This will also affect the application flow.

Initializing the USART Driver

Describes how to initialize the USART Driver.

Description

Before the USART driver can be opened, it must be configured and initialized. The driver build time configuration is defined by the configuration macros. Refer to the Building the Library section for the location of and more information on the various configuration macros and how these macros should be designed. The driver initialization is configured through the DRV_USART_INIT data structure that is passed to the DRV_USART_Initialize function. The initialization parameters include the USART baud, the USART peripheral, USART interrupts and read queue and write queue sizes (which are applicable only when buffer queue data transfer is used). The following code shows an example of initializing the USART driver for 300 bps and uses USART2. If the driver is configured for Interrupt mode of operation, the application should set the priority of USART interrupts.

```
/* The following code shows an example of designing the
 * DRV_USART_INIT data structure. It also shows how an example
 * usage of the DRV_USART_Initialize() function and how Interrupt
 * System Service routines are used to set USART Interrupt
 * priority. */

  DRV_USART_INIT usartInit;
  SYS_MODULE_OBJ usartModule1;

  /* Set the baud to 300 */
  usartInit.baud = 300;

  /* Auto Baud detection or Stop Idle is not needed */
  usartInit.flags = DRV_USART_INIT_FLAG_NONE;

  /* Handshaking is not needed */
  usartInit.handshake = DRV_USART_HANDSHAKE_NONE;

  /* USART Error Interrupt source for this USART
   * driver instance. Note that INT_SOURCE_USART_2_ERROR
   * value is defined by the Interrupt System Service and
   * is the error interrupt for USART 2*/
  usartInit.interruptError = INT_SOURCE_USART_2_ERROR;

  /* USART Receive Interrupt source for this USART
   * driver instance. Note that INT_SOURCE_USART_2_RECEIVE
   * value is defined by the Interrupt System Service and
   * is the error interrupt for USART 2 */
  usartInit.interruptReceive = INT_SOURCE_USART_2_RECEIVE;

  /* USART Transmit Interrupt source for this USART
   * driver instance. Note that INT_SOURCE_USART_2_TRANSMIT
   * value is defined by the Interrupt System Service and
   * is the error interrupt for USART 2 */
  usartInit.interruptTransmit = INT_SOURCE_USART_2_TRANSMIT;

  /* Line control mode */
  usartInit.lineControl = DRV_USART_LINE_CONTROL_8NONE1;

  /* Operation mode is normal. Loopback or addressed is not
   * needed */
  usartInit.mode = DRV_USART_OPERATION_MODE_NORMAL;

  /* Peripheral Bus clock frequency at which the USART is
   * operating */
  usartInit.brgClock = 80000000;

  /* System module power setting. Typically set to
   * SYS_MODULE_POWER_RUN_FULL */
  usartInit.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;

  /* Receive buffer queue size. In this case a maximum of 2
   * receive buffers can be queued. Only applicable if the
```

```
 * Buffer Queue Data Transfer Model is included in the
 * application. */
usartInit.queueSizeReceive = 2;

/* Transmit buffer queue size. In this case a maximum of 3
 * transmit buffers can be queued. Only applicable if the
 * Buffer Queue Data Transfer Model is included in the
 * application. */
usartInit.queueSizeTransmit = 3;

/* The USART peripheral instance index associated with this
 * driver instance. Note that this value is defined by the
 * USART Peripheral Library */
usartInit.usartID = USART_ID_2;

/* Initialize USART Driver Instance 0 */
usartModule1 = DRV_USART_Initialize(DRV_USART_0, (SYS_MODULE_INIT*)&usartInit);

/* The result of the driver initialization can be checked */
if(SYS_MODULE_OBJ_INVALID == usartModule1)
{
    /* There was an error in initialization. */
}

/* If the USART driver is configured for interrupt mode of
 * operation, the interrupt priorities should be configured.
 * Here the Interrupt System Service is used to set the
 * priority to level 4 */

/* Initialize the interrupt system service */
SYS_INT_Initialize();

/* Set the USART 2 module interrupt priority to 4*/
SYS_INT_VectorPrioritySet(INT_VECTOR_UART2, INT_PRIORITY_LEVEL4);

/* Set the USART 2 module interrupt sub priority to 0*/
SYS_INT_VectorSubprioritySet(INT_VECTOR_UART2, INT_SUBPRIORITY_LEVEL0);

/* Enable global interrupt */
SYS_INT_Enable();
```

The USART Driver can be configured to transfer data through the DMA. This requires the application to specify the DMA channels to be used for USART transmit and receive operations. The USART Driver depends on the DMA System Service to access the DMA module. The DMA channels to be used for transmit and receive transfers should be specified in the DRV_USART_INIT data structure. The usage of DMA channels for transmit operations is independent of the usage of DMA channels for receive operations. It is therefore possible to configure the USART Driver to use a DMA channel for transmit operation without using it for receive operation and vice versa. The USART Driver Interrupt mode (configured by the DRV_USART_INTERRUPT_MODE macro) only affects the transfer direction that does not use DMA. The following code shows an example of using the USART driver initialization to use DMA for transferring data. The code also shows example initialization of the DMA System Service.

```
/* The following code shows an example of designing the
 * DRV_USART_INIT data structure. It also shows how an example
 * usage of the DRV_USART_Initialize() function and how Interrupt
 * System Service routines are used to set USART Interrupt
 * priority. */

  DRV_USART_INIT usartInit;
  SYS_DMA_INIT dmaInit;
  SYS_MODULE_OBJ usartModule1;
  SYS_MODULE_OBJ dmaModule;

  /* Set the baud to 300 */
  usartInit.baud = 300;

  /* Auto Baud detection or Stop Idle is not needed */
  usartInit.flags = DRV_USART_INIT_FLAG_NONE;
```

```
/* Handshaking is not needed */
usartInit.handshake = DRV_USART_HANDSHAKE_NONE;

/* USART Error Interrupt source for this USART
 * driver instance. Note that INT_SOURCE_USART_2_ERROR
 * value is defined by the Interrupt System Service and
 * is the error interrupt for USART2*/
usartInit.interruptError = INT_SOURCE_USART_2_ERROR;

/* USART Receive Interrupt source for this USART
 * driver instance. Note that INT_SOURCE_USART_2_RECEIVE
 * value is defined by the Interrupt System Service and
 * is the receive interrupt for USART2 */
usartInit.interruptReceive = INT_SOURCE_USART_2_RECEIVE;

/* USART Transmit Interrupt source for this USART
 * driver instance. Note that INT_SOURCE_USART_2_TRANSMIT
 * value is defined by the Interrupt System Service and
 * is the transmit interrupt for USART2 */
usartInit.interruptTransmit = INT_SOURCE_USART_2_TRANSMIT;

/* Line control mode */
usartInit.lineControl = DRV_USART_LINE_CONTROL_8NONE1;

/* Operation mode is normal. Loopback or addressed is not
 * needed */
usartInit.mode = DRV_USART_OPERATION_MODE_NORMAL;

/* Peripheral Bus clock frequency at which the USART is
 * operating */
usartInit.brgClock = 80000000;

/* System module power setting. Typically set to
 * SYS_MODULE_POWER_RUN_FULL */
usartInit.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;

/* Receive buffer queue size. In this case a maximum of 2
 * receive buffers can be queued. Only applicable if the
 * Buffer Queue Data Transfer Model is included in the
 * application. */
usartInit.queueSizeReceive = 2;

/* Transmit buffer queue size. In this case a maximum of 3
 * transmit buffers can be queued. Only applicable if the
 * Buffer Queue Data Transfer Model is included in the
 * application. */
usartInit.queueSizeTransmit = 3;

/* The USART peripheral instance index associated with this
 * driver instance. Note that this value is defined by the
 * USART Peripheral Library */
usartInit.usartID = USART_ID_2;

/* Use DMA channel 1 for transmit. If transmit via DMA is
 * not required, set this to DMA_CHANNEL_NONE. These values
 * are defined by the DMA System Service. */
usartInit.dmaChannelTransmit = DMA_CHANNEL_1;

/* Use DMA channel 2 for receive. If receive via DMA is
 * not required, set this to DMA_CHANNEL_NONE. These values
 * are defined by the DMA System Service. */
usartInit.dmaChannelReceive = DMA_CHANNEL_2;

/* Set the interrupt source for the Transmit DMA channel.
 * This parameter is ignored if the dmaChannelTransmit
 * parameter is set to DMA_CHANNEL_NONE. */
usartInit.dmaInterruptTransmit = INT_SOURCE_DMA_1;

/* Set the interrupt source for the Receive DMA channel.
```

```
 * This parameter is ignored if the dmaChannelReceive
 * parameter is set to DMA_CHANNEL_NONE. */
usartInit.dmaInterruptReceive = INT_SOURCE_DMA_2;

/********* End of DRV_USART_INIT Initialization *************/


/* If the USART driver is configured for interrupt mode of
 * operation, the interrupt priorities should be configured.
 * Here the Interrupt System Service is used to set the
 * priority to level 4 */

/* Initialize the interrupt system service */
SYS_INT_Initialize();

/* Set the USART 2 module interrupt priority to 4*/
SYS_INT_VectorPrioritySet(INT_VECTOR_UART2, INT_PRIORITY_LEVEL4);

/* Set the USART 2 module interrupt sub priority to 0*/
SYS_INT_VectorSubprioritySet(INT_VECTOR_UART2, INT_SUBPRIORITY_LEVEL0);

/* Set the DMA 1 channel interrupt priority to 4*/
SYS_INT_VectorPrioritySet(INT_VECTOR_DMA1, INT_PRIORITY_LEVEL4);

/* Set the DMA 1 channel interrupt sub priority to 0*/
SYS_INT_VectorSubprioritySet(INT_VECTOR_DMA1, INT_SUBPRIORITY_LEVEL0);

/* Set the DMA 2 channel interrupt priority to 4*/
SYS_INT_VectorPrioritySet(INT_VECTOR_DMA2, INT_PRIORITY_LEVEL4);

/* Set the DMA 2 channel interrupt sub priority to 0*/
SYS_INT_VectorSubprioritySet(INT_VECTOR_DMA2, INT_SUBPRIORITY_LEVEL0);

/* Enable global interrupt */
SYS_INT_Enable();

/* This is the DMA System Service Initialization */
dmaInit.sidl = SYS_DMA_SIDL_DISABLE;
dmaModule = SYS_DMA_Initialize((SYS_MODULE_INIT*)&dmaInit);

/* The result of the DMA System Service initialization can be checked */
if(SYS_MODULE_OBJ_INVALID == dmaModule)
{
    /* DMA System Service initialization was not successful */
}

/* Initialize USART Driver Instance 0 */
usartModule1 = DRV_USART_Initialize(DRV_USART_0, (SYS_MODULE_INIT*)&usartInit);

/* The result of the driver initialization can be checked */
if(SYS_MODULE_OBJ_INVALID == usartModule1)
{
    /* There was an error in initialization. */
}
```

Opening the USART Driver

Describes how to open the USART Driver.

Description

To use the USART driver, the application must open the driver. This is done by calling the DRV_USART_Open function. Calling this function with DRV_IO_INTENT_NONBLOCKING will cause the driver to be opened in non blocking mode. The DRV_USART_Read and DRV_USART_Write functions when called by this client will be non blocking. . Calling this function with DRV_IO_INTENT_BLOCKING will cause the driver to be opened in blocking mode. The DRV_USART_Read and DRV_USART_Write functions when called by this client will be blocking.

If successful, the DRV_USART_Open function will return a handle to the driver. This handle records the association between the client and the driver instance that was opened. The DRV_USART_Open function may return DRV_HANDLE_INVALID in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return an invalid handle in other (error) cases as well.

The following code shows an example of the driver being opened in different modes.

```
DRV_HANDLE usartHandle1, usartHandle2;

/* Client 1 opens the USART driver in non blocking mode */
usartHandle1 = DRV_USART_Open(DRV_USART_0, DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_NONBLOCKING);

/* Check if the handle is valid */
if(DRV_HANDLE_INVALID == usartHandle1)
{
    /* The driver was not opened successfully. The client
     * can try opening it again */
}

/* Client 2 opens the USART driver in blocking mode */
usartHandle2 = DRV_USART_Open(DRV_USART_0, DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_BLOCKING);

/* Check if the handle is valid */
if(DRV_HANDLE_INVALID == usartHandle2)
{
    /* The driver was not opened successfully. The client
     * can try opening it again */
}

/* The client can also open the USART driver in read only mode
 * (DRV_IO_INTENT_READ), write only mode (DRV_IO_INTENT_WRITE)
 * and exclusive mode (DRV_IO_INTENT_EXCLUSIVE). If the driver
 * has been opened exclusively by a client, it cannot be opened
 * again by another client */
```

Byte Data Transfer Model

Describes the USART Driver byte data transfer model.

Description

To use the Byte Data Transfer model, the DRV_USART_BYTE_MODEL_SUPPORT configuration macro should be true. The `drv_usart_byte_model.c` function should be included in the application project. The application cannot support the Read/Write and Buffer Queue Data transfer model when the Byte Model is enabled.

The following code shows an example of how the DRV_USART_WriteByte function and the DRV_USART_ReadByte function are used.

```
 /* Client uses the a byte model API to write a byte*/
if(!DRV_USART_TransmitBufferIsFull(usartHandle1))
{
    byte = '1';
    DRV_USART_WriteByte(usartHandle1,byte);
}

/* Client waits until data is available and then reads
 * byte */
 while(DRV_USART_ReceiverBufferIsEmpty(usartHandle1));
        byte = DRV_USART_ReadByte(usartHandle1);
```

File I/O Type Read Write/Data Transfer Model

This topic describes the File I/O Type Read Write Data Transfer .

Description

To use the File I/O Type Read Write Data Transfer Model, the DRV_USART_READ_WRITE_MODEL_SUPPORT configuration macro should be 'true'. The file `drv_usart_read_write.c` file should be included in the application project. The driver can support the non-DMA Buffer Queue Data Transfer Model along with the File I/O Type Read Write Data Transfer Model. The Byte-by-Byte Model and DMA Buffer Queue Model cannot be enabled if the File I/O Type Read Write Data Transfer Model is enabled.

The DRV_USART_Read and DRV_USART_Write function represent the File I/O Type Read Write Data Transfer Model. The functional behavior of these API is affected by the mode in which the client opened the driver. If the client opened the driver in blocking mode, then these API will block. In blocking mode, the DRV_USART_Read and DRV_USART_Write functions will not return until the requested number of bytes have been read or written. If the client opened the driver in non-blocking mode, then these API will not block. In non-blocking mode, the DRV_USART_Read and DRV_USART_Write functions will return immediately with the amount of data that could be read or written.

Note: Do not open the driver in Blocking mode when the driver is configured for polling operation (DRV_USART_INTERRUPT_MODE is false) in a bare-metal (non RTOS) application. This will cause the system to enter an infinite loop condition when the DRV_USART_Read or DRV_USART_Write function is called.

The following code shows an example of File I/O Type Read Write Data Transfer Model usage when the driver is opened in Blocking mode.

```
/* This code shows the functionality of the DRV_USART_Write and
 * DRV_USART_Read function when the driver is opened in blocking mode */

DRV_HANDLE usartHandle1;
uint8_t myData[10];
size_t bytesProcessed;

/* The driver is opened in blocking mode */
usartHandle1 = DRV_USART_Open(DRV_USART_0, DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_BLOCKING);

/* Check if the driver was opened successfully */
if(DRV_HANDLE_INVALID == usartHandle1)
{
    /* The driver could not be opened successfully */
}

/* Transmit 10 bytes from the myData array. Function will not return until 10 bytes
 * have been accepted by the driver. This is because the client opened the driver
 * in blocking mode. */

bytesProcessed = DRV_USART_Write(usartHandle1, myData, 10);

/* Read 10 bytes from the myData array. Function will not return until all 10 bytes
 * have been received by the driver. This is because the client opened the driver
 * in blocking mode. */

bytesProcessed = DRV_USART_Read(usartHandle1, myData, 10);
```

In non-Blocking mode, the driver uses the internal USART Hardware FIFO as storage. The DRV_USART_Read function checks if bytes are available in USART Receive Hardware FIFO. If bytes are available, these are read and the number of bytes read is returned. The DRV_USART_Write function checks if USART Transmit Hardware FIFO has empty location. If locations are empty, the bytes to be transmitted are queued up in the FIFO and the number of queued bytes is returned. In either case, the number of bytes read or written may be less than the number requested by the client. The client can, in such a case, call the DRV_USART_Read and/or the DRV_USART_Write functions again to process the pending bytes. The following code shows how this can be done.

```
/* This code shows the functionality of the DRV_USART_Write and
 * DRV_USART_Read functions when the driver is opened in non-blocking mode */
```

```
DRV_HANDLE usartHandle1;
uint8_t myData[10];
size_t bytesProcessed;

/* The driver is opened in non-blocking mode */
usartHandle1 = DRV_USART_Open(DRV_USART_0,
        DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_NONBLOCKING);

/* Check if the driver was opened successfully */
if(DRV_HANDLE_INVALID == usartHandle1)
{
    /* The driver could not be opened successfully */
}

/* The following code call the DRV_USART_Write function
 * multiple times to write 10 bytes completely. Note how the
 * function return value is used to update the location of
 * user source data. */

bytesProcessed = 0;
do
{
    /* Write data to the USART and use the return value to
     * update the source data pointer and pending bytes number. */
    bytesProcessed += DRV_USART_Write(usartHandle1,
            myData + bytesProcessed, (10 - bytesProcessed));

} while(bytesProcessed < 10);

/* The following code calls the DRV_USART_Read function multiple times to read
 * 10 bytes completely. Note how the function return value is used to update the
 * location of user destination array. */

bytesProcessed = 0;
do
{
    /* Read data from the USART and use the return value to update the
     * destination pointer and pending bytes number. */
    bytesProcessed += DRV_USART_Read(usartHandle1,
            myData + bytesProcessed, (10 - bytesProcessed));

}while (bytesProcessed < 10);
```

Buffer Queue Transfer Model

This topic describes the Buffer Queue Data Transfer Model.

Description

To use the Buffer Queue Data Transfer Model, the DRV_USART_BUFFER_QUEUE_SUPPORT configuration macro should be true. The file, `drv_usart_buffer_queue.c`, should be included in the application project. If the DMA-enabled Buffer Queue Model is required, the `drv_usart_buffer_queue_dma.c` file (*and not* the `drv_usart_buffer_queue.c`) should be included in the application project. The DMA and non-DMA Buffer Queue model API is the same. The driver can support the non-DMA Buffer Queue Data Transfer Model along with the File I/O Type Read Write Data Transfer Model. The Byte by Byte Model cannot be enabled if the Buffer Queue Data Transfer Model is enabled.

The DRV_USART_BufferAddRead and DRV_USART_BufferAddWrite functions represent the Buffer Queue Data Transfer Model. These functions are always non-blocking. The Buffer Queue Data Transfer Model employs queuing of read and write request. Each driver instance contains a read and write queue. The size of the read queue is determined by the queueSizeRead member of the DRV_USART_INIT data structure. The size of the write queue is determined by the queueSizeWrite member of the DRV_USART_INIT data structure. The driver provides driver events (DRV_USART_BUFFER_EVENT) that indicates termination of the buffer requests.

When the driver is configured for Interrupt mode operation, the buffer event handler executes in an interrupt context. Calling computationally intensive or hardware polling routines within the event handlers is not recommended.

When the driver adds request to the queue, it returns a buffer handle. This handle allows the client to track the request at it progresses through the queue. The buffer handle expires when the event associated with the buffer completes. The following code shows an example of using the Buffer Queue Data Transfer Model.

```
/* This code shows an example of using the
 * Buffer Queue Data Transfer Model. */
DRV_HANDLE usartHandle1;
uint8_t myData1[10], myData2[10];
uint8_t myData3[10], myData4[10];
size_t bytesProcessed;
DRV_USART_BUFFER_HANDLE bufferHandle1, bufferHandle2;
DRV_USART_BUFFER_HANDLE bufferHandle3, bufferHandle4;


/* The driver is opened in non blocking mode */
usartHandle1 = DRV_USART_Open(DRV_USART_0,
        DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_NONBLOCKING);

/* Check if the driver was opened successfully */
if(DRV_HANDLE_INVALID == usartHandle1)
{
    /* The driver could not be opened successfully */
}

/* Register a Buffer Event Handler with USART driver.
 * This event handler function will be called whenever
 * there is a buffer event.  An application defined
 * context can also be specified. This is returned when
 * the event handler is called.
 * */
DRV_USART_BufferEventHandlerSet(usartHandle1,
        APP_USARTBufferEventHandler, NULL);


/* Queue up two buffers for transmit */
DRV_USART_BufferAddWrite(usartHandle1, &bufferHandle1, myData1, 10);
DRV_USART_BufferAddWrite(usartHandle1, &bufferHandle2, myData2, 10);

/* Queue up two buffers for receive */
DRV_USART_BufferAddRead(usartHandle1, &bufferHandle3, myData3, 10);
DRV_USART_BufferAddRead(usartHandle1, &bufferHandle4, myData4, 10);

/* This is application USART Driver Buffer Event Handler */
```

```
void APP_USARTBufferEventHandler(DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:
            /* This means the data was transferred */
            break;
        case DRV_USART_BUFFER_EVENT_ERROR:
            /* Error handling here. */
            break;
        default:
            break;
    }
}
```

Driver Tasks Routine

This topic describes the Driver Tasks Routine.

Description

The USART driver contains three task routines, DRV_USART_TasksTransmit, DRV_USART_TasksReceive and DRV_USART_TasksError. These task routines implement the USART Driver state machines for transmit, receive and error related operations. If the driver is configured for polling operation, the required Task routine should be called in SYS_Tasks routine of the application. If the driver is configured for interrupt mode of operation, the task routine should be called from the Interrupt Service Routine. The following code shows an example of both.

```
/* The following code shows an example of
 * USART2 Interrupt Service Routine. This function
 * will be called when a USART2 interrupt occurs
 * and the driver is configured for interrupt mode
 * operation */

void __ISR ( _UART_2_VECTOR,ipl4 ) _InterruptHandler_USART ( void )
{
    /* usartModule1 is the System Module Object
     * that was returned by the DRV_USART_Initialize
     * function. */

    DRV_USART_TasksTransmit(usartModule1);
    DRV_USART_TasksReceive(usartModule1);
    DRV_USART_TasksError(usartModule1);

}

/* In case of Polled mode, the tasks routines are
 * invoked from the SYS_Tasks() routine. */

void SYS_Tasks(void)
{
    DRV_USART_TasksTransmit(usartModule1);
    DRV_USART_TasksReceive(usartModule1);
    DRV_USART_TasksError(usartModule1);
}

/* The SYS_Tasks routine is invoked from the main
 * application while(1) loop. */

while(1)
{
    SYS_Tasks();
}
```

Using the USART Driver with DMA

This topic provides information on using the USART Driver with DMA.

Description

To use the USART Driver with DMA, the following should be noted:

- Include `drv_usart_dma.c` in the project. Do not include `drv_usart.c`.
- Include `drv_usart_buffer_queue_dma.c` in the project. Do not include `drv_usart_buffer_queue.c`.
- Initialize the driver to use DMA. Refer to Initializing the USART Driver for details.
- Refer to the DMA System Service section for details on initializing and using the DMA system service in Polling or Interrupt mode
- The DRV_USART_INTERRUPT_MODE configuration macro should be set to 'true'
- Do not directly invoke the DRV_USART_TasksTransmit and DRV_USART_TasksReceive functions

Configuring the Library

Macros

	Name	Description
	DRV_USART_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_USART_BAUD_RATE	The USART baud rate for static build of the driver.
	DRV_USART_INDEX	USART Static Index selection
	DRV_USART_INTERRUPT_MODE	Macro controls interrupt based operation of the driver
	DRV_USART_INTERRUPT_SOURCE_ERROR	Defines the interrupt source for the error interrupt
	DRV_USART_PERIPHERAL_ID	Configures the USART PLIB Module ID
	DRV_USART_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_USART_BUFFER_QUEUE_SUPPORT	Specifies if the Buffer Queue support should be enabled.
	DRV_USART_BYTE_MODEL_SUPPORT	Specifies if the Byte Model support should be enabled.
	DRV_USART_INTERRUPT_SOURCE_RECEIVE	Macro to define the Receive interrupt source in case of static driver
	DRV_USART_INTERRUPT_SOURCE_RECEIVE_DMA	Macro to define the Receive DMA Channel interrupt source in case of static driver
	DRV_USART_INTERRUPT_SOURCE_TRANSMIT	Macro to define the Transmit interrupt source in case of static driver
	DRV_USART_INTERRUPT_SOURCE_TRANSMIT_DMA	Macro to define the Transmit DMA Channel interrupt source in case of static driver
	DRV_USART_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.
	DRV_USART_READ_WRITE_MODEL_SUPPORT	Specifies if the Read Write Model support should be enabled.
	DRV_USART_RECEIVE_DMA	Macro to defines the USART Driver Receive DMA Channel in case of static driver
	DRV_USART_TRANSMIT_DMA	Macro to defines the USART Driver Transmit DMA Channel in case of static driver

Description

The USART Driver requires the specification of compile-time configuration macros. These macros define resource usage, feature availability, and dynamic behavior of the driver. These configuration macros should be defined in the `system_config.h` file.

This header can be placed anywhere in the application specific folders and the path of this header needs to be presented to the include search for a successful build. Refer to the Applications Overview section for more details.

Note: Initialization overrides are not supported in this version.

```
/* In this configuration example, the USART driver
 * must manage only on USART peripheral instance.
 * This macro can be greater than one if more
 * USART peripherals are needed. Not defining this
 * macro will cause the driver to be built in
 * static mode */
#define DRV_USART_INSTANCES_NUMBER 1
```

```
/* There will be 3 different client that use the
 * one instance of the USART peripheral. Note that
 * this macro configures the total (combined) number of clients
 * across all instance of the USART driver. Not defining
 * this macro will cause the driver to be configured
 * for single client operation */
#define DRV_USART_CLIENTS_NUMBER 3

/* USART Driver should be built for interrupt mode.
 * Set this to false for Polled mode operation */
#define DRV_USART_INTERRUPT_MODE true

/* Combined buffer queue depth is 5. Refer to the
 * description of the Buffer Queue data transfer model
 * and the DRV_USART_QUEUE_DEPTH_COMBINED macro
 * for more details on how this is configured. */
#define DRV_USART_QUEUE_DEPTH_COMBINED 5

/* Set this macro to true is Buffer Queue data
 * transfer model is to be enabled. */
#define DRV_USART_BUFFER_QUEUE_SUPPORT true

/* Set this macro to true if Byte by Byte data
 * transfer model is to be enabled. */
#define DRV_USART_BYTE_MODEL_SUPPORT false

/* Set this macro to true File IO type Read Write
 * data transfer model is to be enabled */
#define DRV_USART_READ_WRITE_MODEL_SUPPORT false
```

DRV_USART_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_usart_config_template.h

C

```
#define DRV_USART_CLIENTS_NUMBER 4
```

Description

USART Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. This value represents the total number of clients to be supported across all hardware instances. So if USART1 will be accessed by 2 clients and USART2 will accessed by 3 clients, then this number should be 5. It is recommended that this be set exactly equal to the number of expected clients. Client support consumes RAM memory space. If this macro is not defined and the DRV_USART_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - single client operation. If this macro is defined and the DRV_USART_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - multi client operation.

Remarks

None

DRV_USART_BAUD_RATE Macro

The USART baud rate for static build of the driver.

File

drv_usart_config_template.h

C

```
#define DRV_USART_BAUD_RATE 9600
```

Description

USART Baud Rate Control

The USART baud rate for static build of the driver. When the driver is built for static mode, this macro overrides the baud rate member in the driver initialization data structure.

Remarks

None

DRV_USART_INDEX Macro

USART Static Index selection

File

drv_usart_config_template.h

C

```
#define DRV_USART_INDEX DRV_USART_INDEX_2
```

Description

Index - Used for static drivers

USART Static Index selection for the driver object reference. This macro defines the driver index in case of static and static multi-client build. For example, if this macro is set to DRV_USART_INDEX_2, then static driver APIs would be DRV_USART2_Initialize(), DRV_USART2_Open() etc. When building static drivers, this macro should be different for each static build of the USART driver that needs to be included in the project.

Remarks

This index is required to make a reference to the driver object

DRV_USART_INTERRUPT_MODE Macro

Macro controls interrupt based operation of the driver

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_MODE true
```

Description

USART Interrupt Mode Operation Control

This macro controls the interrupt based operation of the driver. The possible values it can take are

- true - Enables the interrupt mode
- false - Enables the polling mode

If the macro value is true, then Interrupt Service Routine for the interrupt should be defined in the application. The DRV_USART_Tasks() routine should be called in the ISR. While using the USART driver with DMA, this flag should

always be true.

Remarks

None

DRV_USART_INTERRUPT_SOURCE_ERROR Macro

Defines the interrupt source for the error interrupt

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_SOURCE_ERROR INT_SOURCE_USART_2_ERROR
```

Description

Error Interrupt Source

Macro to define the Error interrupt source in case of static driver. The interrupt source defined by this macro will override the errorInterruptSource member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the USART module error interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None

DRV_USART_PERIPHERAL_ID Macro

Configures the USART PLIB Module ID

File

drv_usart_config_template.h

C

```
#define DRV_USART_PERIPHERAL_ID USART_ID_2
```

Description

USART Peripheral Library Module ID

This macro configures the PLIB ID if the driver is built statically. This value will override the usartID member of the DRV_USART_INIT initialization data structure. In that when the driver is built statically, the usartID member of the DRV_USART_INIT data structure will be ignored by the driver initialization routine and this macro will be considered. This should be set to the PLIB ID of USART module (USART_ID_1, USART_ID_2 and so on).

Remarks

None

DRV_USART_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_usart_config_template.h

C

```
#define DRV_USART_INSTANCES_NUMBER 2
```

Description

USART driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of USART modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None

DRV_USART_BUFFER_QUEUE_SUPPORT Macro

Specifies if the Buffer Queue support should be enabled.

File

drv_usart_config_template.h

C

```
#define DRV_USART_BUFFER_QUEUE_SUPPORT true
```

Description

USART Driver Buffer Queue Support

This macro defines if Buffer Queue support should be enabled. Setting this macro to true will enable buffer queue support and all buffer related driver function. The driver should be built along with the drv_usart_buffer_queue.c file which contains the functional implementation for buffer queues. If buffer queue operation is enabled, then DRV_USART_BYTE_MODEL_SUPPORT should not be true. If this macro is set to false, the behavior of the USART Driver Buffer Queue API is not defined. While using the USART driver with DMA, the driver supports Buffer Queue Data transfer model irrespective of the value of this configuration macro.

Remarks

None

DRV_USART_BYTE_MODEL_SUPPORT Macro

Specifies if the Byte Model support should be enabled.

File

drv_usart_config_template.h

C

```
#define DRV_USART_BYTE_MODEL_SUPPORT false
```

Description

USART Driver Byte Model Support

This macro defines if Byte Model support should be enabled. Setting this macro to true will enable byte model support and all byte operation related driver functions. The driver should be built along with the drv_usart_byte_model.c file which contains the functional implementation for byte model operation. If byte model operation is enabled, then driver will not support buffer queue and read write models. The behaviour of the byte mode API when this macro is set to false is not defined.

Remarks

None

DRV_USART_INTERRUPT_SOURCE_RECEIVE Macro

Macro to define the Receive interrupt source in case of static driver

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_SOURCE_RECEIVE INT_SOURCE_USART_2_RECEIVE
```

Description

Receive Interrupt Source

Macro to define the Receive interrupt source in case of static driver. The interrupt source defined by this macro will override the rxInterruptSource member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the USART module receive interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_USART_INTERRUPT_SOURCE_RECEIVE_DMA Macro

Macro to define the Receive DMA Channel interrupt source in case of static driver

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_SOURCE_RECEIVE_DMA
```

Description

Receive DMA Channel Interrupt Source

Macro to define the Receive DMA Channel interrupt source in case of static driver. The interrupt source defined by this macro will override the dmaInterruptReceive member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_USART_INTERRUPT_SOURCE_TRANSMIT Macro

Macro to define the Transmit interrupt source in case of static driver

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_SOURCE_TRANSMIT INT_SOURCE_USART_2_TRANSMIT
```

Description

Transmit Interrupt Source

Macro to define the TX interrupt source in case of static driver. The interrupt source defined by this macro will override the txInterruptSource member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the USART module transmit interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_USART_INTERRUPT_SOURCE_TRANSMIT_DMA Macro

Macro to define the Transmit DMA Channel interrupt source in case of static driver

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_SOURCE_TRANSMIT_DMA
```

Description

Transmit DMA Channel Interrupt Source

Macro to define the TX DMA Channel interrupt source in case of static driver. The interrupt source defined by this macro will override the dmaInterruptTransmit member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_USART_QUEUE_DEPTH_COMBINED Macro

Number of entries of all queues in all instances of the driver.

File

drv_usart_config_template.h

C

```
#define DRV_USART_QUEUE_DEPTH_COMBINED 16
```

Description

USART Driver Instance combined queue depth.

This macro defines the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for transmit and receive operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build). The hardware instance transmit buffer queue will queue transmit buffers submitted by the DRV_USART_BufferAddWrite() function. The hardware instance receive buffer queue will queue receive buffers submitted by the DRV_USART_BufferAddRead() function.

A buffer queue will contain buffer queue entries, each related to a BufferAdd request. This configuration macro

defines total number of buffer entries that will be available for use between all USART driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking read and write requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its transmit and receive buffer queue size.

As an example, consider the case of static single client driver application where full duplex non blocking operation is desired without queuing, the minimum transmit queue depth and minimum receive queue depth should be 1. Hence the total number of buffer entries should be 2.

As an example, consider the case of a dynamic driver (say 2 instances) where instance 1 will queue up to 3 write requests and up to 2 read requests, and instance 2 will queue up to 2 write requests and up to 6 read requests, the value of this macro should be 13 (2 + 3 + 2 + 6).

Remarks

None

DRV_USART_READ_WRITE_MODEL_SUPPORT Macro

Specifies if the Read Write Model support should be enabled.

File

drv_usart_config_template.h

C

```
#define DRV_USART_READ_WRITE_MODEL_SUPPORT true
```

Description

USART Driver Read Write Model Support

This macro defines if Read Write Model support should be enabled. Setting this macro to true will enable read write model support and all read write related driver functions. The driver should be built along with the drv_usart_read_write.c file which contains the functional implementation for byte model operation. If read write model operation is enabled, then DRV_USART_BYTE_MODEL_SUPPORT should not be true. The behaviour of the Read Write Model API when this macro is set to false is not defined.

Remarks

None

DRV_USART_RECEIVE_DMA Macro

Macro to defines the USART Driver Receive DMA Channel in case of static driver

File

drv_usart_config_template.h

C

```
#define DRV_USART_RECEIVE_DMA
```

Description

USART Driver Receive DMA Channel

Macro to define the USART Receive DMA Channel in case of static driver. The DMA channel defined by this macro

will override the dmaReceive member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel in the DMA PLIB for the microcontroller.

Remarks

None.

DRV_USART_TRANSMIT_DMA Macro

Macro to defines the USART Driver Transmit DMA Channel in case of static driver

File

drv_usart_config_template.h

C

```
#define DRV_USART_TRANSMIT_DMA
```

Description

USART Driver Transmit DMA Channel

Macro to define the USART Transmit DMA Channel in case of static driver. The DMA channel defined by this macro will override the dmaTransmit member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel in the DMA PLIB for the microcontroller.

Remarks

None.

Building the Library

This section lists the files that are available in the USART Driver Library.

Description

This section list the files that are available in the \src folder of the USART Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/usart`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_usart.h`	This file should be included by any .c file which accesses the USART Driver API. This one file contains the prototypes for all driver API.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_usart.c`	This file should always be included in the project when using the USART Driver.
`/src/dynamic/drv_usart_dma.c`	This file should always be included in the project when using the USART driver with DMA.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
`/src/dynamic/drv_usart_byte_model.c`	This file should be included in the project if the USART Driver Byte Model API is required.
`/src/dynamic/drv_usart_buffer_queue.c`	This file should be included in the project if the USART Driver Buffer Queue Model API (without DMA) is required.
`/src/dynamic/drv_usart_read_write.c`	This file should be included in the project if the USART Driver Read Write Model API is required.
`/src/dynamic/drv_usart_buffer_queue_dma.c`	This file should be included in the project if the USART Driver Buffer Queue Model API with DMA is required.

Module Dependencies

The USART Driver Library depends on the following modules:

- Interrupt System Service Library
- DMA System Service Library (if USART Driver is configured to use DMA)

Library Interface

a) System Functions

	Name	Description
	DRV_USART_Initialize	Initializes the USART instance for the specified driver index. **Implementation:** Static/Dynamic
	DRV_USART_Deinitialize	Deinitializes the specified instance of the USART driver module. **Implementation:** Static/Dynamic
	DRV_USART_Status	Gets the current status of the USART driver module. **Implementation:** Dynamic
	DRV_USART_TasksReceive	Maintains the driver's receive state machine and implements its ISR. **Implementation:** Dynamic
	DRV_USART_TasksTransmit	Maintains the driver's transmit state machine and implements its ISR. **Implementation:** Dynamic
	DRV_USART_TasksError	Maintains the driver's error state machine and implements its ISR. **Implementation:** Dynamic

b) Core Client Functions

	Name	Description
	DRV_USART_Open	Opens the specified USART driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_USART_Close	Closes an opened-instance of the USART driver. **Implementation:** Dynamic
	DRV_USART_ClientStatus	Gets the current client-specific status the USART driver. **Implementation:** Dynamic
	DRV_USART_ErrorGet	This function returns the error(if any) associated with the last client request. **Implementation:** Dynamic

c) Communication Management Client Functions

	Name	Description
	DRV_USART_BaudSet	This function changes the USART module baud to the specified value. **Implementation:** Dynamic
	DRV_USART_LineControlSet	This function changes the USART module line control to the specified value. **Implementation:** Dynamic

d) Buffer Queue Read/Write Client Functions

	Name	Description
	DRV_USART_BufferAddRead	Schedule a non-blocking driver read operation. **Implementation:** Dynamic
	DRV_USART_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_USART_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_USART_BufferProcessedSizeGet	This function returns number of bytes that have been processed for the specified buffer. **Implementation:** Dynamic

e) File I/O Type Read/Write Functions

	Name	Description
	DRV_USART_Read	Reads data from the USART. **Implementation:** Dynamic
	DRV_USART_Write	Writes data to the USART. **Implementation:** Dynamic

f) Byte Transfer Functions

	Name	Description
	DRV_USART_ReadByte	Reads a byte of data from the USART. **Implementation:** Static/Dynamic
	DRV_USART_WriteByte	Writes a byte of data to the USART. **Implementation:** Static/Dynamic
	DRV_USART_TransmitBufferSizeGet	Returns the size of the transmit buffer. **Implementation:** Dynamic
	DRV_USART_ReceiverBufferSizeGet	Returns the size of the receive buffer. **Implementation:** Dynamic
	DRV_USART_TransferStatus	Returns the transmitter and receiver transfer status. **Implementation:** Dynamic
	DRV_USART_TransmitBufferIsFull	Provides the status of the driver's transmit buffer. **Implementation:** Dynamic
	DRV_USART_ReceiverBufferIsEmpty	Provides the status of the driver's receive buffer. **Implementation:** Static/Dynamic

g) Data Types and Constants

	Name	Description
	DRV_USART_CLIENT_STATUS	Defines the client-specific status of the USART driver.
	DRV_USART_INIT	Defines the data required to initialize or reinitialize the USART driver
	DRV_USART_INIT_FLAGS	Flags identifying features that can be enabled when the driver is initialized.
	DRV_USART_TRANSFER_STATUS	Specifies the status of the receive or transmit
	DRV_USART_INDEX_0	USART driver index definitions
	DRV_USART_INDEX_1	This is macro DRV_USART_INDEX_1.
	DRV_USART_INDEX_2	This is macro DRV_USART_INDEX_2.
	DRV_USART_INDEX_3	This is macro DRV_USART_INDEX_3.
	DRV_USART_INDEX_4	This is macro DRV_USART_INDEX_4.
	DRV_USART_INDEX_5	This is macro DRV_USART_INDEX_5.
	DRV_USART_BAUD_SET_RESULT	Identifies the handshaking modes supported by the USART driver.
	DRV_USART_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_USART_BUFFER_EVENT_HANDLER	Pointer to a USART Driver Buffer Event handler function
	DRV_USART_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the driver.
	DRV_USART_ERROR	Defines the possible errors that can occur during driver operation.
	DRV_USART_LINE_CONTROL_SET_RESULT	Identifies the results of the baud set function.

	DRV_USART_OPERATION_MODE_DATA	Defines the initialization data required for different operation modes of USART.
	DRV_USART_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_USART_COUNT	Number of valid USART drivers
	DRV_USART_READ_ERROR	USART Driver Read Error.
	DRV_USART_WRITE_ERROR	USART Driver Write Error.
	DRV_USART_LINE_CONTROL	Identifies the line control modes supported by the USART driver.
	DRV_USART_OPERATION_MODE	Identifies the modes of the operation of the USART module

Description

This section describes the functions of the USART Driver Library.

Refer to each section for a detailed description.

a) System Functions

DRV_USART_Initialize Function

Initializes the USART instance for the specified driver index.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
SYS_MODULE_OBJ DRV_USART_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const
init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the USART driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the USART module ID. For example, driver instance 0 can be assigned to USART2. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_USART_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other USART routine is called.

This routine should only be called once during system initialization unless DRV_USART_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
// The following code snippet shows an example USART driver initialization.
// The driver is initialized for normal mode and a baud of 300. The
// receive queue size is set to 2 and transmit queue size is set to 3.

DRV_USART_INIT              usartInit;
SYS_MODULE_OBJ              objectHandle;

usartInit.baud  = 300;
usartInit.mode  = DRV_USART_OPERATION_MODE_NORMAL;
usartInit.flags = DRV_USART_INIT_FLAG_NONE;
usartInit.usartID   = USART_ID_2;
usartInit.brgClock  = 80000000;
usartInit.handshake = DRV_USART_HANDSHAKE_NONE;
usartInit.lineControl       = DRV_USART_LINE_CONTROL_8NONE1;
usartInit.interruptError    = INT_SOURCE_USART_2_ERROR;
usartInit.interruptReceive  = INT_SOURCE_USART_2_RECEIVE;
usartInit.queueSizeReceive  = 2;
usartInit.queueSizeTransmit = 3;
usartInit.interruptTransmit = INT_SOURCE_USART_2_TRANSMIT;
usartInit.moduleInit.value  = SYS_MODULE_POWER_RUN_FULL;

objectHandle = DRV_USART_Initialize(DRV_USART_INDEX_1, (SYS_MODULE_INIT*)&usartInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

```
SYS_MODULE_OBJ DRV_USART_Initialize
(
const SYS_MODULE_INDEX index,
const SYS_MODULE_INIT * const init
)
```

DRV_USART_Deinitialize Function

Deinitializes the specified instance of the USART driver module.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
void DRV_USART_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the USART driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_USART_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_USART_Initialize
SYS_STATUS          status;

DRV_USART_Deinitialize(object);

status = DRV_USART_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_USART_Initialize routine

Function

void DRV_USART_Deinitialize(SYS_MODULE_OBJ object)

DRV_USART_Status Function

Gets the current status of the USART driver module.

Implementation: Dynamic

File

drv_usart.h

C

```
SYS_STATUS DRV_USART_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

Description

This routine provides the current status of the USART driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_USART_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USART_Initialize
SYS_STATUS          usartStatus;

usartStatus = DRV_USART _Status(object);
if (SYS_STATUS_READY == usartStatus)
{
    // This means the driver can be opened using the
    // DRV_USART_Open() function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_USART_Initialize routine

Function

SYS_STATUS DRV_USART_Status(SYS_MODULE_OBJ object)

DRV_USART_TasksReceive Function

Maintains the driver's receive state machine and implements its ISR.

Implementation: Dynamic

File

drv_usart.h

C

```
void DRV_USART_TasksReceive(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal receive state machine and implement its receive ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks function. In interrupt mode, this function should be called in the receive interrupt service routine of the USART that is associated with this USART driver hardware instance.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USART_Initialize

while (true)
{
    DRV_USART_TasksReceive (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_USART_Initialize)

Function

void DRV_USART_TasksReceive (SYS_MODULE_OBJ object);

DRV_USART_TasksTransmit Function

Maintains the driver's transmit state machine and implements its ISR.

Implementation: Dynamic

File

drv_usart.h

C

```
void DRV_USART_TasksTransmit(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal transmit state machine and implement its transmit ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks function. In interrupt mode, this function should be called in the transmit interrupt service routine of the USART that is associated with this USART driver hardware instance.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USART_Initialize

while (true)
{
    DRV_USART_TasksTransmit (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_USART_Initialize)

Function

void DRV_USART_TasksTransmit (SYS_MODULE_OBJ object);

DRV_USART_TasksError Function

Maintains the driver's error state machine and implements its ISR.

Implementation: Dynamic

File

drv_usart.h

C

```
void DRV_USART_TasksError(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal error state machine and implement its error ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks function. In interrupt mode, this function should be called in the error interrupt service routine of the USART that is associated with this USART driver hardware instance.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USART_Initialize

while (true)
{
    DRV_USART_TasksError (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_USART_Initialize)

Function

void DRV_USART_TasksError (SYS_MODULE_OBJ object);

b) Core Client Functions

DRV_USART_Open Function

Opens the specified USART driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_usart.h

C

```
DRV_HANDLE DRV_USART_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_USART_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver is not ready to be opened, typically when the initialize routine has not completed execution.

Description

This routine opens the specified USART driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options additionally affect the behavior of the DRV_USART_Read and DRV_USART_Write functions. If the ioIntent is DRV_IO_INTENT_NONBLOCKING, then these function will not block even if the required amount of data could not be processed. If the ioIntent is DRV_IO_INTENT_BLOCKING, these functions will block until the required amount of data is processed. If the driver is configured for polling and bare-metal operation, it will not support DRV_IO_INTENT_BLOCKING. The driver will operation will always be non-blocking.

If ioIntent is DRV_IO_INTENT_READ, the client will only be able to read from the driver. If ioIntent is DRV_IO_INTENT_WRITE, the client will only be able to write to the driver. If the ioIntent is DRV_IO_INTENT_READWRITE, the client will be able to do both, read and write.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_USART_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application.

Preconditions

Function DRV_USART_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_USART_Open(DRV_USART_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

```
DRV_HANDLE DRV_USART_Open
(
const SYS_MODULE_INDEX index,
const       DRV_IO_INTENT ioIntent
)
```

DRV_USART_Close Function

Closes an opened-instance of the USART driver.

Implementation: Dynamic

File

drv_usart.h

C

```
void DRV_USART_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This routine closes an opened-instance of the USART driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines (with one possible exception described in the "Remarks" section). A new handle must be obtained by calling DRV_USART_Open before the caller may use the driver again

Remarks

Usually there is no need for the client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called. However, if it requires additional time to do so in a non-blocking environment, it will still return from the Close operation but the handle is now a zombie handle. The client can only call the DRV_USART_ClientStatus on a zombie handle to track the completion of the Close operation. The DRV_USART_ClientStatus routine will return DRV_CLIENT_STATUS_CLOSED when the close operation has completed.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_USART_Open

DRV_USART_Close(handle);

// After this point, the handle cannot be used with any other function
// except the DRV_USART_ClientStatus function, which can be used to query
// the success status of the DRV_USART_Close function.

while(DRV_USART_CLIENT_STATUS_CLOSED != DRV_USART_ClientStatus(handle));
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_USART_Close(DRV_Handle handle)

DRV_USART_ClientStatus Function

Gets the current client-specific status the USART driver.

Implementation: Dynamic

File

drv_usart.h

C

```
DRV_USART_CLIENT_STATUS DRV_USART_ClientStatus(DRV_HANDLE handle);
```

Returns

A DRV_USART_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the USART driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize function must have been called.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE              handle;  // Returned from DRV_USART_Open
DRV_USART_CLIENT_STATUS   status;

status = DRV_USART_ClientStatus(handle);
if( DRV_USART_CLIENT_STATUS_CLOSED != status )
{
    // The client had not closed.
}
```

Parameters

Parameters	Description
handle	Handle returned from the driver's open function.

Function

DRV_USART_CLIENT_STATUS DRV_USART_ClientStatus(DRV_HANDLE handle)

DRV_USART_ErrorGet Function

This function returns the error(if any) associated with the last client request.

Implementation: Dynamic

File

drv_usart.h

C

```
DRV_USART_ERROR DRV_USART_ErrorGet(const DRV_HANDLE client);
```

Returns

A DRV_USART_ERROR type indicating last known error status.

Description

This function returns the error(if any) associated with the last client request. DRV_USART_Read and DRV_USART_Write will update the client error status when these functions return DRV_USART_TRANSFER_ERROR. If the driver send a DRV_USART_BUFFER_EVENT_ERROR to the client, the client can call this function to know the error cause. The error status will be updated on every operation and should be read frequently (ideally immediately after the driver operation has completed) to know the relevant error status.

Remarks

It is the client's responsibility to make sure that the error status is obtained frequently. The driver will update the client error status regardless of whether this has been examined by the client. This function is thread safe when used in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver. This is done once.

DRV_USART_BufferEventHandlerSet( myUSARTHandle, APP_USARTBufferEventHandle,
                                 (uintptr_t)&myAppObj );

bufferHandle = DRV_USART_BufferAddRead( myUSARThandle,
                                        myBuffer, MY_BUFFER_SIZE );

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_USARTBufferEventHandler( DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle )
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
```

```
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_SUCCESS:

            // This means the data was transferred.
            break;

        case DRV_USART_BUFFER_EVENT_FAILURE:

            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred. We can also find
            // the error cause.

            processedBytes = DRV_USART_BufferProcessedSizeGet(bufferHandle);
            if(DRV_USART_ERROR_RECEIVE_OVERRUN == DRV_USART_ErrorGet(myUSARTHandle))
            {
                // There was an receive over flow error.
                // Do error handling here.
            }

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
bufferhandle	Handle of the buffer of which the processed number of bytes to be obtained.

Function

DRV_USART_ERROR DRV_USART_ErrorGet(DRV_HANDLE client);

c) Communication Management Client Functions

DRV_USART_BaudSet Function

This function changes the USART module baud to the specified value.

Implementation: Dynamic

File

drv_usart.h

C

```
DRV_USART_BAUD_SET_RESULT DRV_USART_BaudSet(const DRV_HANDLE client, uint32_t baud);
```

Returns

None.

Description

This function changes the USART module baud to the specified value. Any queued buffer requests will be processed at the updated baud. The USART driver operates at the baud specified in DRV_USART_Initialize function unless the DRV_USART_BaudSet function is called to change the baud.

Remarks

The implementation of this function, in this release of the driver, changes the baud immediately. This may interrupt on-going data transfer. It is recommended that the driver be opened exclusively if this function is to be called. This function is thread safe when used in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

DRV_USART_BaudSet(myUSARTHandle, 9600);
```

Parameters

Parameters	Description
handle	client handle returned by DRV_USART_Open function.
baud	desired baud.

Function

void DRV_USART_BaudSet(DRV_HANDLE client, uint32_t baud);

DRV_USART_LineControlSet Function

This function changes the USART module line control to the specified value.

Implementation: Dynamic

File

drv_usart.h

C

```
DRV_USART_LINE_CONTROL_SET_RESULT DRV_USART_LineControlSet(const DRV_HANDLE client, const
DRV_USART_LINE_CONTROL lineControl);
```

Returns

DRV_USART_LINE_CONTROL_SET_SUCCESS if the function was successful. Returns DRV_HANDLE_INVALID if the client handle is not valid.

Description

This function changes the USART module line control parameters to the specified value. Any queued buffer requests will be processed at the updated line control parameters. The USART driver operates at the line control parameters specified in DRV_USART_Initialize function unless the DRV_USART_LineControlSet function is called to change the line control parameters.

Remarks

The implementation of this function, in this release of the driver, changes the line control immediately. This may interrupt on-going data transfer. It is recommended that the driver be opened exclusively if this function is to be called. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

DRV_USART_LineControlSet(myUSARTHandle, DRV_USART_LINE_CONTROL_8NONE1);
```

Parameters

Parameters	Description
handle	client handle returned by DRV_USART_Open function.
lineControl	line control parameters.

Function

void DRV_USART_LineControlSet(DRV_HANDLE client,
DRV_USART_LINE_CONTROL lineControl);

d) Buffer Queue Read/Write Client Functions

DRV_USART_BufferAddRead Function

Schedule a non-blocking driver read operation.

Implementation: Dynamic

File

drv_usart.h

C

```
void DRV_USART_BufferAddRead(const DRV_HANDLE handle, DRV_USART_BUFFER_HANDLE * const
bufferHandle, void * buffer, const size_t size);
```

Returns

The buffer handle is returned in the bufferHandle argument. This is DRV_USART_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_USART_BUFFER_HANDLE_INVALID in the bufferHandle argument:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient.
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_USART_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_USART_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the USART Driver Buffer Event Handler that is registered by the client. It should not be called in the event handler associated with another USART driver instance. It should not be called directly in an ISR.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART device instance and the DRV_USART_Status must have returned SYS_STATUS_READY.

DRV_USART_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_USART_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_USART_BUFFER_HANDLE bufferHandle;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver

DRV_USART_BufferEventHandlerSet(myUSARTHandle,
                APP_USARTBufferEventHandler, (uintptr_t)&myAppObj);

DRV_USART_BufferAddRead(myUSARThandle, &bufferHandle,
```

```
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_USARTBufferEventHandler(DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_USART_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_USART_Open function.
buffer	Buffer where the received data will be stored.
size	Buffer size in bytes.

Function

void DRV_USART_BufferAddRead

(

const DRV_HANDLE handle,

DRV_USART_BUFFER_HANDLE * bufferHandle,

void * buffer,

const size_t size

)

DRV_USART_BufferAddWrite Function

Schedule a non-blocking driver write operation.

Implementation: Dynamic

File

drv_usart.h

C

```
void DRV_USART_BufferAddWrite(const DRV_HANDLE handle, DRV_USART_BUFFER_HANDLE * bufferHandle,
void * buffer, const size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_USART_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. On returning, the bufferHandle parameter may be DRV_USART_BUFFER_HANDLE_INVALID for the following reasons:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read-only
- if the buffer size is 0
- if the transmit queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_USART_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or a DRV_USART_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the USART Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another USART driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART device instance and the DRV_USART_Status must have returned SYS_STATUS_READY.

DRV_USART_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_USART_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver

DRV_USART_BufferEventHandlerSet(myUSARTHandle,
                APP_USARTBufferEventHandler, (uintptr_t)&myAppObj);

DRV_USART_BufferAddWrite(myUSARThandle, &bufferHandle,
```

```
                                        myBuffer, MY_BUFFER_SIZE);

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_USARTBufferEventHandler(DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_USART_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the communication channel as return by the DRV_USART_Open function.
bufferHandle	Pointer to an argument that will contain the return buffer handle.
buffer	Data to be transmitted.
size	Buffer size in bytes.

Function

void DRV_USART_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_USART_BUFFER_HANDLE * bufferHandle,

void * buffer,

size_t size

);

DRV_USART_BufferEventHandlerSet Function

Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Implementation: Dynamic

File

drv_usart.h

C

```
void DRV_USART_BufferEventHandlerSet(const DRV_HANDLE handle, const
DRV_USART_BUFFER_EVENT_HANDLER eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls either the DRV_USART_BufferAddRead or DRV_USART_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_USART_BUFFER_HANDLE bufferHandle;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver. This is done once

DRV_USART_BufferEventHandlerSet( myUSARTHandle, APP_USARTBufferEventHandle,
                                 (uintptr_t)&myAppObj );

DRV_USART_BufferAddRead(myUSARThandle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.
```

```
void APP_USARTBufferEventHandler(DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_SUCCESS:

            // This means the data was transferred.
            break;

        case DRV_USART_BUFFER_EVENT_FAILURE:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_USART_BufferEventHandlerSet

(

const DRV_HANDLE handle,

const DRV_USART_EVENT_HANDLER eventHandler,

const uintptr_t context

)

DRV_USART_BufferProcessedSizeGet Function

This function returns number of bytes that have been processed for the specified buffer.

Implementation: Dynamic

File

drv_usart.h

C

```
size_t DRV_USART_BufferProcessedSizeGet(DRV_USART_BUFFER_HANDLE bufferHandle);
```

Returns

Returns the number of the bytes that have been processed for this buffer.

Returns 0 for an invalid or an expired buffer handle.

Description

This function returns number of bytes that have been processed for the specified buffer. The client can use this function, in a case where the buffer has terminated due to an error, to obtain the number of bytes that have been processed. This function can be used for non-DMA buffer transfers only. It cannot be used when the USART driver is configured to use DMA.

If this function is called on a invalid buffer handle, or if the buffer handle has expired, then the function returns 0.

Remarks

This function is thread safe when used in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Either the DRV_USART_BufferAddRead or DRV_USART_BufferAddWrite function must have been called and a valid buffer handle returned.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver. This is done once

DRV_USART_BufferEventHandlerSet( myUSARTHandle, APP_USARTBufferEventHandle,
                                 (uintptr_t)&myAppObj );

bufferHandle = DRV_USART_BufferAddRead( myUSARThandle,
                                        myBuffer, MY_BUFFER_SIZE );

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_USARTBufferEventHandler( DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle )
{
```

```
        // The context handle was set to an application specific
        // object. It is now retrievable easily in the event handler.
        MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
        size_t processedBytes;

        switch(event)
        {
            case DRV_USART_BUFFER_EVENT_SUCCESS:

                // This means the data was transferred.
                break;

            case DRV_USART_BUFFER_EVENT_FAILURE:

                // Error handling here.
                // We can find out how many bytes were processed in this
                // buffer before the error occurred.

                processedBytes = DRV_USART_BufferProcessedSizeGet(bufferHandle);

                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
bufferhandle	Handle of the buffer of which the processed number of bytes to be obtained.

Function

size_t DRV_USART_BufferProcessedSizeGet(DRV_USART_BUFFER_HANDLE bufferHandle);

e) File I/O Type Read/Write Functions

DRV_USART_Read Function

Reads data from the USART.

Implementation: Dynamic

File

drv_usart.h

C

```
size_t DRV_USART_Read(const DRV_HANDLE handle, void * buffer, const size_t numbytes);
```

Returns

Number of bytes actually copied into the caller's buffer. Returns DRV_USART_READ_ERROR in case of an error.

Description

This routine reads data from the USART. This function is blocking if the driver was opened by the client for blocking operation. This function will not block if the driver was opened by the client for non blocking operation. If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_BLOCKING, this function will only return when (or will block until) numbytes of bytes have been received or if an error occurred. If there are buffers queued for receiving data, these buffers will be serviced first. The function will not return until the requested number of bytes have been read.

If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_NON_BLOCKING, this function will return with the number of bytes that were actually read. The function will not wait until numBytes of bytes have been read. If there are buffer queued for reading data, then the function will not block and will return immediately with 0 bytes read.

Remarks

This function is thread safe in a RTOS application. If the driver is configured for polled operation, this it will not support blocking operation in a bare metal (non-RTOS) application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_USART_Open call.

Example

```
DRV_HANDLE       myUSARTHandle;    // Returned from DRV_USART_Open
char             myBuffer[MY_BUFFER_SIZE];
unsigned int     count;
unsigned int     total;

total = 0;
do
{
    count  = DRV_USART_Read(myUSARTHandle, &myBuffer[total], MY_BUFFER_SIZE - total);
    if(count == DRV_USART_READ_ERROR)
    {
       // There was an error. The DRV_USART_ErrorGet() function
       // can be called to find the exact error.
    }
    total += count;

    // Do something else...

} while( total < MY_BUFFER_SIZE );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
buffer	Buffer into which the data read from the USART instance will be placed.
numbytes	Total number of bytes that need to be read from the module instance (must be equal to or less than the size of the buffer)

Function

```
size_t DRV_USART_Read
(
const    DRV_HANDLE handle,
void * buffer,
const size_t numbytes
)
```

© 2013-2015 Microchip Technology Inc.

DRV_USART_Write Function

Writes data to the USART.

Implementation: Dynamic

File

drv_usart.h

C

```
size_t DRV_USART_Write(const DRV_HANDLE handle, void * buffer, const size_t numbytes);
```

Returns

Number of bytes actually written to the driver. Return DRV_USART_WRITE_ERROR in case of an error.

Description

This routine writes data to the USART. This function is blocking if the driver was opened by the client for blocking operation. This function will not block if the driver was opened by the client for non blocking operation. If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_BLOCKING, this function will only return when (or will block until) numbytes of bytes have been transmitted or if an error occurred. If there are buffers queued for writing, the function will wait until all the preceding buffers are completed. Ongoing buffer transmit operations will not be affected.

If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_NON_BLOCKING, this function will return with the number of bytes that were actually accepted for transmission. The function will not wait until numBytes of bytes have been transmitted. If there a buffers queued for transmit, the function will not wait and will return immediately with 0 bytes.

Remarks

This function is thread safe in a RTOS application. This function is thread safe in a RTOS application. If the driver is configured for polled operation, this it will not support blocking operation in a bare metal (non-RTOS) application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_USART_Open call.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
char            myBuffer[MY_BUFFER_SIZE];
int     count;
unsigned int    total;

total = 0;
do
{
    count  = DRV_USART_Write(myUSARTHandle, &myBuffer[total],
                                        MY_BUFFER_SIZE - total);
    total += count;

    // Do something else...

} while( total < MY_BUFFER_SIZE );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
buffer	Buffer containing the data to written.

numbytes	size of the buffer

Function

```
size_t DRV_USART_Write( const DRV_HANDLE handle, void * buffer,
const size_t numbytes )
```

f) Byte Transfer Functions

DRV_USART_ReadByte Function

Reads a byte of data from the USART.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
uint8_t DRV_USART_ReadByte(const DRV_HANDLE handle);
```

Returns

A data byte received by the driver.

Description

This routine reads a byte of data from the USART.

Remarks

This function is thread safe when called in a RTOS application. Note that DRV_USART_WriteByte and DRV_USART_ReadByte function cannot co-exist with DRV_USART_BufferAddRead, DRV_USART_BufferAddWrite, DRV_USART_Read and DRV_USART_Write functions in a application. Calling the DRV_USART_ReadByte and DRV_USART_WriteByte functions will disrupt the processing of any queued buffers.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

The transfer status should be checked to see if the receiver is not empty before calling this function.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
char            myBuffer[MY_BUFFER_SIZE];
unsigned int    numBytes;

numBytes = 0;
do
{
    if( DRV_USART_TRANSFER_STATUS_RECEIVER_DATA_PRESENT & DRV_USART_TransferStatus(myUSARTHandle) )
    {
        myBuffer[numBytes++] = DRV_USART_ReadByte(myUSARTHandle);
    }

    // Do something else...

} while( numBytes < MY_BUFFER_SIZE);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint8_t DRV_USART_ReadByte(const DRV_HANDLE handle)

DRV_USART_WriteByte Function

Writes a byte of data to the USART.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
void DRV_USART_WriteByte(const DRV_HANDLE handle, const uint8_t byte);
```

Returns

None.

Description

This routine writes a byte of data to the USART.

Remarks

This function is thread safe when called in a RTOS application. Note that DRV_USART_WriteByte and DRV_USART_ReadByte function cannot co-exist with DRV_USART_BufferAddRead, DRV_USART_BufferAddWrite, DRV_USART_Read and DRV_USART_Write functions in a application. Calling the DRV_USART_ReadByte and DRV_USART_WriteByte function will disrupt the processing of any queued buffers.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

The transfer status should be checked to see if transmitter is not full before calling this function.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
char            myBuffer[MY_BUFFER_SIZE];
unsigned int    numBytes;

// Preinitialize myBuffer with MY_BUFFER_SIZE bytes of valid data.

numBytes = 0;
while( numBytes < MY_BUFFER_SIZE );
{
    if( !(DRV_USART_TRANSFER_STATUS_TRANSMIT_FULL & DRV_USART_TransferStatus(myUSARTHandle)) )
    {
        DRV_USART_WriteByte(myUSARTHandle, myBuffer[numBytes++]);
    }

    // Do something else...
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
byte	Data byte to write to the USART

Function

void DRV_USART_WriteByte(const DRV_HANDLE handle, const uint8_t byte)

DRV_USART_TransmitBufferSizeGet Function

Returns the size of the transmit buffer.

Implementation: Dynamic

File

drv_usart.h

C

```
unsigned int DRV_USART_TransmitBufferSizeGet(const DRV_HANDLE handle);
```

Returns

Size of the driver's transmit buffer, in bytes.

Description

This routine returns the size of the transmit buffer and can be used by the application to determine the number of bytes to write with the DRV_USART_WriteByte function.

Remarks

Does not account for client queued buffers. This function is thread safe when used in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
const uint8_t   writeBuffer[5];
unsigned int    size, numBytes = 0;
unsigned int    writeBufferLen = sizeof(writeBuffer);

size     = DRV_USART_TransmitBufferSizeGet (myUSARTHandle);

// Do something based on the transmitter buffer size
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

unsigned int DRV_USART_TransmitBufferSizeGet (const DRV_HANDLE handle)

DRV_USART_ReceiverBufferSizeGet Function

Returns the size of the receive buffer.

Implementation: Dynamic

File

drv_usart.h

C

```
unsigned int DRV_USART_ReceiverBufferSizeGet(const DRV_HANDLE handle);
```

Returns

Size of the driver's receive buffer, in bytes.

Description

This routine returns the size of the receive buffer.

Remarks

Does not account for client queued buffers. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
const uint8_t   readBuffer[5];
unsigned int    size, numBytes = 0;
unsigned int    readbufferLen = sizeof(readBuffer);

size     = DRV_USART_ReceiverBufferSizeGet(myUSARTHandle);

// Do something based on the receiver buffer size
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

unsigned int DRV_USART_ReceiverBufferSizeGet(const DRV_HANDLE handle)

DRV_USART_TransferStatus Function

Returns the transmitter and receiver transfer status.

Implementation: Dynamic

File

drv_usart.h

C

```
DRV_USART_TRANSFER_STATUS DRV_USART_TransferStatus(const DRV_HANDLE handle);
```

Returns

A DRV_USART_TRANSFER_STATUS value describing the current status of the transfer.

Description

This returns the transmitter and receiver transfer status.

Remarks

The returned status may contain a value with more than one of the bits specified in the DRV_USART_TRANSFER_STATUS enumeration set. The caller should perform an "AND" with the bit of interest and verify if the result is non-zero (as shown in the example) to verify the desired status bit. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open

if (DRV_USART_TRANSFER_STATUS_RECEIVER_DATA_PRESENT & DRV_USART_TransferStatus(myUSARTHandle))
{
    // Data has been received that can be read
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_USART_TRANSFER_STATUS DRV_USART_TransferStatus(const DRV_HANDLE handle)

DRV_USART_TransmitBufferIsFull Function

Provides the status of the driver's transmit buffer.

Implementation: Dynamic

File

drv_usart.h

C

```
bool DRV_USART_TransmitBufferIsFull(const DRV_HANDLE handle);
```

Returns

true - if the transmit buffer is full

false - if the transmit buffer is not full

Description

This routine identifies if the driver's transmit buffer is full or not. This function can be used in conjunction with the DRV_USART_Write and DRV_USART_WriteByte functions.

Remarks

Does not account for client queued buffers. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
unsigned int    numBytes;
int             bytesToWrite;
const uint8_t   writeBuffer[35] = "1234567890ABCDEFGHIJKLMNOPn" ;
int             writebufferLen = strlen((char *)writeBuffer);

numBytes = 0;
while( numBytes < writebufferLen )
{
    if (DRV_USART_TransmitBufferisFull())
    {
        // Do something else until there is some room in the driver's Transmit buffer.
    }
    else
    {
        DRV_USART_WriteByte(myUSARTHandle, writeBuffer[numBytes++]);
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

bool DRV_USART_TransmitBufferIsFull(const DRV_HANDLE handle)

DRV_USART_ReceiverBufferIsEmpty Function

Provides the status of the driver's receive buffer.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
bool DRV_USART_ReceiverBufferIsEmpty(const DRV_HANDLE handle);
```

Returns

true - if the driver's receive buffer is empty

false - if the driver's receive buffer is not empty

Description

This routine indicates if the driver's receiver buffer is empty. This function can be used in conjunction with the DRV_USART_Read and DRV_USART_ReadByte functions.

Remarks

Does not account for client queued buffers. This function is safe thread safe when used in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE                myUSARTHandle;     // Returned from DRV_USART_Open
char                      myBuffer[MY_BUFFER_SIZE];
unsigned int              numBytes;

numBytes = 0;
while( numBytes < MY_BUFFER_SIZE );
{
    if ( !DRV_USART_ReceiverBufferIsEmpty(myUSARTHandle) )
    {
        if( numBytes < MY_BUFFER_SIZE )
        {
            myBuffer[numBytes++] = DRV_USART_ReadByte (myUSARTHandle);
        }
        else
        {
            break;
        }
    }

    // Do something else while more data is received.
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

bool DRV_USART_ReceiverBufferIsEmpty(const DRV_HANDLE handle)

g) Data Types and Constants

DRV_USART_CLIENT_STATUS Type

Defines the client-specific status of the USART driver.

File

drv_usart.h

C

```
typedef enum DRV_USART_CLIENT_STATUS@1 DRV_USART_CLIENT_STATUS;
```

Description

USART Client-Specific Driver Status

This enumeration defines the client-specific status codes of the USART driver.

Remarks

Returned by the DRV_USART_ClientStatus function.

DRV_USART_INIT Type

Defines the data required to initialize or reinitialize the USART driver

File

drv_usart.h

C

```
typedef struct DRV_USART_INIT@1 DRV_USART_INIT;
```

Description

USART Driver Initialization Data

This data type defines the data required to initialize or reinitialize the USART driver. If the driver is built statically, the members of this data structure are statically over-ridden by static override definitions in the system_config.h file.

Remarks

None.

DRV_USART_INIT_FLAGS Type

Flags identifying features that can be enabled when the driver is initialized.

File

drv_usart.h

C

```
typedef enum DRV_USART_INIT_FLAGS@1 DRV_USART_INIT_FLAGS;
```

Description

USART Initialization flags

This enumeration defines flags identifying features that can be enabled when the driver is initialized.

Remarks

These flags can be logically ORed together. They are passed into the DRV_USART_Initialize function through the "flags" member of the DRV_USART_INIT structure.

DRV_USART_TRANSFER_STATUS Type

Specifies the status of the receive or transmit

File

drv_usart.h

C

```
typedef enum DRV_USART_TRANSFER_STATUS@1 DRV_USART_TRANSFER_STATUS;
```

Description

USART Driver Transfer Flags

This type specifies the status of the receive or transmit operation.

Remarks

More than one of these values may be OR'd together to create a complete status value. To test a value of this type, the bit of interest must be ANDed with the value and checked to see if the result is non-zero.

DRV_USART_INDEX_0 Macro

USART driver index definitions

File

drv_usart.h

C

```
#define DRV_USART_INDEX_0 0
```

Description

Driver USART Module Index

These constants provide USART driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_USART_Initialize and DRV_USART_Open routines to identify the driver instance in use.

DRV_USART_INDEX_1 Macro

File

drv_usart.h

C

```
#define DRV_USART_INDEX_1 1
```

Description

This is macro DRV_USART_INDEX_1.

DRV_USART_INDEX_2 Macro

File

drv_usart.h

C

```
#define DRV_USART_INDEX_2 2
```

Description

This is macro DRV_USART_INDEX_2.

DRV_USART_INDEX_3 Macro

File

drv_usart.h

C

```
#define DRV_USART_INDEX_3 3
```

Description

This is macro DRV_USART_INDEX_3.

DRV_USART_INDEX_4 Macro

File

drv_usart.h

C

```
#define DRV_USART_INDEX_4 4
```

Description

This is macro DRV_USART_INDEX_4.

DRV_USART_INDEX_5 Macro

File

drv_usart.h

C

```
#define DRV_USART_INDEX_5 5
```

Description

This is macro DRV_USART_INDEX_5.

DRV_USART_BAUD_SET_RESULT Enumeration

Identifies the handshaking modes supported by the USART driver.

File

drv_usart.h

C

```
typedef enum {
} DRV_USART_BAUD_SET_RESULT;
```

Description

USART Handshake Modes

This data type identifies the handshaking modes supported by the USART driver.

Remarks

Not all modes are available on all devices. Refer to the specific device data sheet to determine availability.

DRV_USART_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_usart.h

C

```
typedef enum {
  DRV_USART_BUFFER_EVENT_COMPLETE,
  DRV_USART_BUFFER_EVENT_ERROR,
  DRV_USART_BUFFER_EVENT_ABORT
} DRV_USART_BUFFER_EVENT;
```

Members

Members	Description
DRV_USART_BUFFER_EVENT_COMPLETE	All data from or to the buffer was transferred successfully.
DRV_USART_BUFFER_EVENT_ERROR	There was an error while processing the buffer transfer request.
DRV_USART_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

USART Driver Buffer Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_USART_BufferAddRead or DRV_USART_BufferAddWrite functions.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_USART_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_USART_BUFFER_EVENT_HANDLER Type

Pointer to a USART Driver Buffer Event handler function

File

drv_usart.h

C

```
typedef void (* DRV_USART_BUFFER_EVENT_HANDLER)(DRV_USART_BUFFER_EVENT event,
DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t context);
```

Returns

None.

Description

USART Driver Buffer Event Handler Function Pointer

This data type defines the required function signature for the USART driver buffer event handling callback function. A client must register a pointer to a buffer event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values and are described here and a partial example implementation is provided.

Remarks

If the event is DRV_USART_BUFFER_EVENT_COMPLETE, it means that the data was transferred successfully.

If the event is DRV_USART_BUFFER_EVENT_ERROR, it means that the data was not transferred successfully. The DRV_USART_ErrorGet function can be called to know the error. The DRV_USART_BufferProcessedSizeGet function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_USART_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The DRV_USART_BufferAddRead and DRV_USART_BufferAddWrite functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running. For example, buffers cannot be added USART2 driver in USART1 driver event handler.

Example

```
void APP_MyBufferEventHandler( DRV_USART_BUFFER_EVENT event,
                               DRV_USART_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_USART_BUFFER_EVENT_ERROR:
        default:

            // Handle error.
            break;
```

```
        }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the vent relates
context	Value identifying the context of the application that registered the event handling function.

DRV_USART_BUFFER_HANDLE Type

Handle identifying a read or write buffer passed to the driver.

File

drv_usart.h

C

```
typedef uintptr_t DRV_USART_BUFFER_HANDLE;
```

Description

USART Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_USART_BufferAddRead or DRV_USART_BufferAddWrite functions. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_USART_ERROR Enumeration

Defines the possible errors that can occur during driver operation.

File

drv_usart.h

C

```
typedef enum {
  DRV_USART_ERROR_ADDRESS
} DRV_USART_ERROR;
```

Members

Members	Description
DRV_USART_ERROR_ADDRESS	Channel address error (Applicable in DMA mode)

Description

USART Driver Errors.

This data type defines the possible errors that can occur when occur during USART driver operation. These values are returned by DRV_USART_ErrorGet function.

Remarks

None

DRV_USART_LINE_CONTROL_SET_RESULT Enumeration

Identifies the results of the baud set function.

File

drv_usart.h

C

```
typedef enum {
} DRV_USART_LINE_CONTROL_SET_RESULT;
```

Description

USART Line Control Set Result

This data type identifies the results of the DRV_USART_LineControlSet function.

Remarks

None.

DRV_USART_OPERATION_MODE_DATA Union

Defines the initialization data required for different operation modes of USART.

File

drv_usart.h

C

```
typedef union {
  struct {
    uint8_t address;
  } AddressedModeInit;
} DRV_USART_OPERATION_MODE_DATA;
```

Members

Members	Description
struct { uint8_t address; } AddressedModeInit;	Initialization for Addressed mode
uint8_t address;	Address of the device.

Description

Operation Mode Initialization Data

This data type defines the initialization data required for different operation modes of the USART.

Remarks

None

DRV_USART_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_usart.h

C

```
#define DRV_USART_BUFFER_HANDLE_INVALID
```

Description

USART Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_USART_BufferAddRead and DRV_USART_BufferAddWrite functions if the buffer add request was not successful.

Remarks

None

DRV_USART_COUNT Macro

Number of valid USART drivers

File

drv_usart.h

C

```
#define DRV_USART_COUNT
```

Description

USART Driver Module Count

This constant identifies the maximum number of USART Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of USART instances on this microcontroller.

Remarks

This value is part-specific.

DRV_USART_READ_ERROR Macro

USART Driver Read Error.

File

drv_usart.h

C

```
#define DRV_USART_READ_ERROR
```

Description

USART Driver Read Error

This constant is returned by DRV_USART_Read() function when an error occurs.

Remarks

None.

DRV_USART_WRITE_ERROR Macro

USART Driver Write Error.

File

drv_usart.h

C

```
#define DRV_USART_WRITE_ERROR
```

Description

USART Driver Write Error

This constant is returned by DRV_USART_Write() function when an error occurs.

Remarks

None.

DRV_USART_LINE_CONTROL Enumeration

Identifies the line control modes supported by the USART driver.

File

drv_usart.h

C

```
typedef enum {
} DRV_USART_LINE_CONTROL;
```

Description

USART Line Control Modes

This data type identifies the line control modes supported by the USART driver. Line control modes define the number of data bits, parity mode, and the number of stop bits in a USART transmit and receive frames.

Remarks

The abbreviations used in the labels for the values of this enumeration follow the format , where:is the number of data bitsis either "NONE" (for no parity), "EVEN" for 1 parity bit added to obtain an even number of bits, or "ODD" for one bit added to obtain an odd number of bits.is the number of Stop bits

DRV_USART_OPERATION_MODE Enumeration

Identifies the modes of the operation of the USART module

File

drv_usart.h

C

```
typedef enum {
  DRV_USART_OPERATION_MODE_IRDA,
  DRV_USART_OPERATION_MODE_NORMAL,
  DRV_USART_OPERATION_MODE_ADDRESSED,
  DRV_USART_OPERATION_MODE_LOOPBACK
} DRV_USART_OPERATION_MODE;
```

Members

Members	Description
DRV_USART_OPERATION_MODE_IRDA	USART works in IRDA mode
DRV_USART_OPERATION_MODE_NORMAL	This is the normal point to point communication mode where the USART communicates directly with another USART by connecting it's Transmit signal to the external USART's Receiver signal and vice versa. An external transceiver may be connected to obtain RS-232 signal levels. This type of connection is typically full duplex.
DRV_USART_OPERATION_MODE_ADDRESSED	This is a multi-point bus mode where the USART can communicate with many other USARTS on a bus using an address-based protocol such as RS-485. This mode is typically half duplex and the physical layer may require a transceiver. In this mode every USART on the bus is assigned an address and the number of data bits is 9 bits
DRV_USART_OPERATION_MODE_LOOPBACK	Loopback mode internally connects the Transmit signal to the Receiver signal, looping data transmission back into this USART's own input. It is useful primarily as a test mode.

Description

USART Modes of Operation

This data type identifies the modes of the operation of the USART module.

Remarks

Not all modes are available on all devices. Refer to the specific device data sheet to determine availability.

Files

Files

Name	Description
drv_usart.h	USART Driver Interface Header File
drv_usart_config_template.h	USART Driver Configuration Template.

Description

This section lists the source and header files used by the USART Driver Library.

drv_usart.h

USART Driver Interface Header File

Enumerations

	Name	Description
	DRV_USART_BAUD_SET_RESULT	Identifies the handshaking modes supported by the USART driver.
	DRV_USART_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_USART_ERROR	Defines the possible errors that can occur during driver operation.
	DRV_USART_LINE_CONTROL	Identifies the line control modes supported by the USART driver.
	DRV_USART_LINE_CONTROL_SET_RESULT	Identifies the results of the baud set function.
	DRV_USART_OPERATION_MODE	Identifies the modes of the operation of the USART module

Functions

	Name	Description
	DRV_USART_BaudSet	This function changes the USART module baud to the specified value. **Implementation:** Dynamic
	DRV_USART_BufferAddRead	Schedule a non-blocking driver read operation. **Implementation:** Dynamic
	DRV_USART_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_USART_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_USART_BufferProcessedSizeGet	This function returns number of bytes that have been processed for the specified buffer. **Implementation:** Dynamic
	DRV_USART_ClientStatus	Gets the current client-specific status the USART driver. **Implementation:** Dynamic
	DRV_USART_Close	Closes an opened-instance of the USART driver. **Implementation:** Dynamic
	DRV_USART_Deinitialize	Deinitializes the specified instance of the USART driver module. **Implementation:** Static/Dynamic

	DRV_USART_ErrorGet	This function returns the error(if any) associated with the last client request. **Implementation:** Dynamic
	DRV_USART_Initialize	Initializes the USART instance for the specified driver index. **Implementation:** Static/Dynamic
	DRV_USART_LineControlSet	This function changes the USART module line control to the specified value. **Implementation:** Dynamic
	DRV_USART_Open	Opens the specified USART driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_USART_Read	Reads data from the USART. **Implementation:** Dynamic
	DRV_USART_ReadByte	Reads a byte of data from the USART. **Implementation:** Static/Dynamic
	DRV_USART_ReceiverBufferIsEmpty	Provides the status of the driver's receive buffer. **Implementation:** Static/Dynamic
	DRV_USART_ReceiverBufferSizeGet	Returns the size of the receive buffer. **Implementation:** Dynamic
	DRV_USART_Status	Gets the current status of the USART driver module. **Implementation:** Dynamic
	DRV_USART_TasksError	Maintains the driver's error state machine and implements its ISR. **Implementation:** Dynamic
	DRV_USART_TasksReceive	Maintains the driver's receive state machine and implements its ISR. **Implementation:** Dynamic
	DRV_USART_TasksTransmit	Maintains the driver's transmit state machine and implements its ISR. **Implementation:** Dynamic
	DRV_USART_TransferStatus	Returns the transmitter and receiver transfer status. **Implementation:** Dynamic
	DRV_USART_TransmitBufferIsFull	Provides the status of the driver's transmit buffer. **Implementation:** Dynamic
	DRV_USART_TransmitBufferSizeGet	Returns the size of the transmit buffer. **Implementation:** Dynamic
	DRV_USART_Write	Writes data to the USART. **Implementation:** Dynamic
	DRV_USART_WriteByte	Writes a byte of data to the USART. **Implementation:** Static/Dynamic

Macros

	Name	Description
	DRV_USART_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_USART_COUNT	Number of valid USART drivers
	DRV_USART_INDEX_0	USART driver index definitions
	DRV_USART_INDEX_1	This is macro DRV_USART_INDEX_1.
	DRV_USART_INDEX_2	This is macro DRV_USART_INDEX_2.
	DRV_USART_INDEX_3	This is macro DRV_USART_INDEX_3.

	DRV_USART_INDEX_4	This is macro DRV_USART_INDEX_4.
	DRV_USART_INDEX_5	This is macro DRV_USART_INDEX_5.
	DRV_USART_READ_ERROR	USART Driver Read Error.
	DRV_USART_WRITE_ERROR	USART Driver Write Error.

Types

	Name	Description
	DRV_USART_BUFFER_EVENT_HANDLER	Pointer to a USART Driver Buffer Event handler function
	DRV_USART_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the driver.
	DRV_USART_CLIENT_STATUS	Defines the client-specific status of the USART driver.
	DRV_USART_INIT	Defines the data required to initialize or reinitialize the USART driver
	DRV_USART_INIT_FLAGS	Flags identifying features that can be enabled when the driver is initialized.
	DRV_USART_TRANSFER_STATUS	Specifies the status of the receive or transmit

Unions

	Name	Description
	DRV_USART_OPERATION_MODE_DATA	Defines the initialization data required for different operation modes of USART.

Description

USART Driver Interface Header File

The USART device driver provides a simple interface to manage the USART or UART modules on Microchip microcontrollers. This file provides the interface definition for the USART driver.

File Name

drv_usart.h

Company

Microchip Technology Inc.

drv_usart_config_template.h

USART Driver Configuration Template.

Macros

	Name	Description
	DRV_USART_BAUD_RATE	The USART baud rate for static build of the driver.
	DRV_USART_BUFFER_QUEUE_SUPPORT	Specifies if the Buffer Queue support should be enabled.
	DRV_USART_BYTE_MODEL_SUPPORT	Specifies if the Byte Model support should be enabled.
	DRV_USART_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_USART_INDEX	USART Static Index selection
	DRV_USART_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported

	DRV_USART_INTERRUPT_MODE	Macro controls interrupt based operation of the driver
	DRV_USART_INTERRUPT_SOURCE_ERROR	Defines the interrupt source for the error interrupt
	DRV_USART_INTERRUPT_SOURCE_RECEIVE	Macro to define the Receive interrupt source in case of static driver
	DRV_USART_INTERRUPT_SOURCE_RECEIVE_DMA	Macro to define the Receive DMA Channel interrupt source in case of static driver
	DRV_USART_INTERRUPT_SOURCE_TRANSMIT	Macro to define the Transmit interrupt source in case of static driver
	DRV_USART_INTERRUPT_SOURCE_TRANSMIT_DMA	Macro to define the Transmit DMA Channel interrupt source in case of static driver
	DRV_USART_PERIPHERAL_ID	Configures the USART PLIB Module ID
	DRV_USART_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.
	DRV_USART_READ_WRITE_MODEL_SUPPORT	Specifies if the Read Write Model support should be enabled.
	DRV_USART_RECEIVE_DMA	Macro to defines the USART Driver Receive DMA Channel in case of static driver
	DRV_USART_TRANSMIT_DMA	Macro to defines the USART Driver Transmit DMA Channel in case of static driver

Description

USART Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_usart_config_template.h

Company

Microchip Technology Inc.

USB Driver Library

This topic describes the USB Driver Library.

Introduction

This library provides an interface to manage the USB module on Microchip family of microcontrollers during different modes of operation.

Description

The USB module implements the physical layer in a USB based system. In that, it allows the USB stack software to access and control the USB. The USB module on Microchip's microcontrollers are compliant with the USB 2.0 specification. This help file only provides relevant information about the operation of the USB. The reader is encouraged to refer to the USB 2.0 specification available at www.usb.org for complete information about USB.

Not all Microchip microcontrollers feature a USB module. Among the ones that do, the supported features may vary. Please refer to the device specific data sheet for details on the availability of the USB module and the supported features.

Note: Trademarks and Intellectual Property are property of their respective owners. Customers are responsible for obtaining appropriate licensing or rights before using this software. Refer to Software License Agreement for complete licensing information.

Using the Library

This topic describes the basic architecture of the USB Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_usb.h`

The interface to the USB Driver library is defined in the `drv_usb.h` header file.

Please refer to the Understanding MPLAB Harmony section for how the Driver interacts with the framework.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the USB module.

Library Interface Section	Description
System Functions	These routines are typically invoked by the MPLAB Harmony system module.
Client Functions	These routines are used by the driver clients to access driver functionality
Pipe Functions	These routines allow clients to manage pipes
Transfer Functions	These routines allow clients to manage transfers
Other Functions	These routines represent the rest of the driver functionality
Data Types and Constants	These data types and constants are required while interacting and configuring the driver

Abstraction Model

This topic describes the abstraction model for the USB Driver Library.

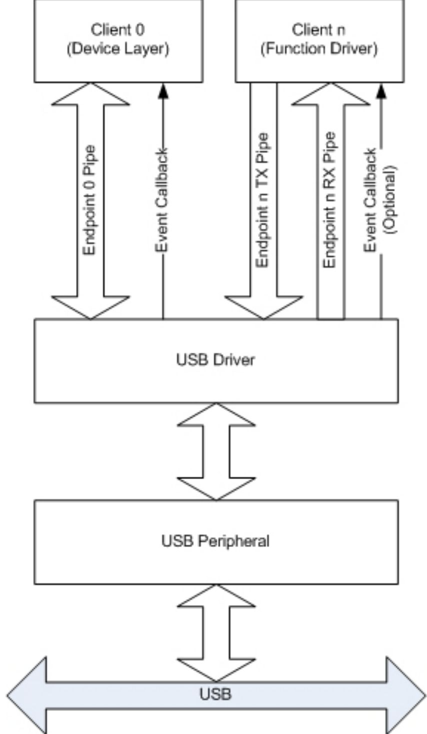
Description

Figure 1 shows the functioning of the USB Driver in device mode. In this mode, the driver offers services required to operate the USB module as a USB device. The USB Driver supports multiple clients. Each client can receive notifications about USB events via callback functions. Typical clients could be the Device Layer firmware and one or more function drivers. The Device layer firmware and the function drivers communicate on the USB via pipes. A pipe is an association between the device's endpoint and host stack software. An Endpoint 0 Pipe is always bidirectional and is a message pipe. Communication on this pipe is defined by the USB specification. All other endpoints support stream pipes. The format of the data on a stream pipe is not specified by the USB specification. Stream pipes are unidirectional. Therefore, two pipes are required to transmit and receive data on a non-zero endpoint.

Clients schedule USB transfers through a pipe. Transfers on non-zero endpoint pipes can be queued. The USB driver manages the transactions required to complete the transfer. This is transparent to the clients. Clients can either get an event notification or can poll to check the completion status of a transfer. Clients can register callback functions to receive USB events notification. A Transfer complete callback function can be registered while requesting a transfer. This function will be called when the transfer is terminated. While operating in device mode, the first client of the USB Driver should always be the USB Device Stack Device layer. Only the first client can schedule control transfers on endpoint 0 (via a pipe) and receive the Setup Token and Setup Handshake events (along with the other events). Other clients may or may not register an event callback function.

The USB Driver uses configuration macros that control among other things, functional aspects such as maximum number of clients and maximum pipes.

Figure 1: USB Driver Device Mode Model



How the Library Works

Before the driver is ready for use, its should be configured. Refer to the Configuring the Library section for more details on how to configure the driver.

To use the USB Driver, initialization and client functions should be invoked in a specific sequence to ensure correct operation. The following is the sequence in which various routines should be called:

1. Call the DRV_USB_Initialize function to initialize the USB Driver. Note that this may be performed by the MPLAB Harmony system module.
2. In USB device mode operation, the first client should always be the USB Device Stack Device Layer. The Device Layer first opens the driver via the DRV_USB_Open function.
3. The Device layer registers an event callback function via the DRV_USB_Device_EventCallBackSet function.
4. The Device layer assigns a setup packet buffer by using the DRV_USB_Device_SetupPacketBufferAssign function. The driver stores the received setup packet in this buffer. Calling this function also enables the USB Device Mode and at this point the device will attach to the host and be ready to receive setup tokens from the host.
5. The Device Layer waits for the Reset Event. In this event, the device layer creates an Endpoint 0 pipe by using the DRV_USB_Device_PipeSetup function. The Device layer can optionally register a transfer complete callback function. This function will be called when a transfer on the endpoint zero is complete.
6. Based on the device enumeration process and the selected configuration, other function drivers may open the driver by using the DRV_USB_Open function.

It is strongly recommended that the sequence previously described be followed for correct operation of the driver.

Note: Not all modes are available on all devices. Please refer to the specific device data sheet for the set of modes supported.

Driver Initialization

At the time of initialization, the DRV_USB_Initialize function requires DRV_MODULE_INIT data structure that provides information required for the function of the driver. The following data is needed:

- Maximum number of endpoints to be supported by the driver. In case endpoints are skipped, the largest endpoint number should be used. For example, if the application uses Endpoints 0, 1 and 2, the maximum number of endpoints should three. If the application uses Endpoints 0, 1 and 5, the maximum number of endpoints should six. The latter configuration is not recommended.
- Maximum side of the Endpoint 0. This should match the maximum size that is reported to the host during enumeration.
- Pointer to byte array whose size is defined DRV_USB_WORKING_BUFFER_SIZE (nEndpoint, maxEP0Size), where nEndpoint is the maximum number of endpoint as described in the first item described previously and maxEP0Size is the maximum endpoint 0 size as previously described in the second item. These values should match usbInitData.ep0MaxSize and usbInitData.nEndpoints.
- The PLIB peripheral ID of the USB module to associated with the driver object being initialized
- The operation mode of the driver (host, device or OTG). Refer to the Release Notes for supported modes.
- The Ping Pong mode to be used by internal endpoint buffers. Refer to the specific device data sheet to determine the supported Ping Pong modes.
- Behavior of the USB module when the microcontroller device enters Idle mode
- Behavior of the USB module when the microcontroller device enters Sleep mode
- Power state of the USB module
- USB module interrupts to be enabled. These include the general, error, and OTG interrupts. Refer to the USB Peripheral Library section for the available module interrupts.
- The interrupt source of the USB module to be used with the driver instance. Refer to the USB Peripheral Library section for the available module interrupts.

The following code shows an example initialization of the driver.

```
/* The following code shows an example initialization of the
 * driver. The USB module to be used in USB1. The maximum number of
 * endpoints to be used is 3 (i.e., endpoint 0,1 and 2). Maximum size of
 * endpoint 0 is 64. The module should not automatically suspend when the
 * micro-controller enters Sleep mode. The module should continue
 * operation when the module enters Idle mode. The ping pong setting
 * is set to full ping pong (both RX and TX endpoints have ping
 * pong buffers). The power state is set to run at full clock speeds.*/

DRV_USB_INIT moduleInit;

uint8_t internalBuffer[[DRV_USB_WORKING_BUFFER_SIZE(3,64)]
                                    __attribute__((aligned(512)));

usbInitData.usbID = USB_ID_1;
usbInitData.usbModuleSetup.OpMode = USB_OPMODE_DEVICE;
usbInitData.usbModuleSetup.ppMode = USB_PING_PONG__FULL_PING_PONG;

usbInitData.usbModuleSetup.StopInIdle       = false;
usbInitData.usbModuleSetup.SuspendInSleep   = false;

usbInitData.usbModuleSetup.otgInterruptEnables = ~USB_OTG_INT_ALL ;
usbInitData.usbModuleSetup.generalInterruptEnables = USB_INT_ALL & ~USB_INT_SOF ;
usbInitData.usbModuleSetup.errorInterruptEnables = USB_ERR_INT_ALL ;

usbInitData.workingBuffer = internalBuffer;
usbInitData.ep0MaxSize = 64;
usbInitData.nEndpoints = 3;
usbInitData.sysModuleInit.powerState = SYS_MODULE_POWER_RUN_FULL ;

usbInitData.interruptSource = INT_SOURCE_USB_1;

/* This is how this data structure is passed to the initialize
 * function. */
```

```
DRV_USB_Initialize(DRV_USB_INDEX_0, (SYS_MODULE_INIT *) &usbInitData);
```

Opening the Driver

Once the driver is initialized for device mode, it can be opened by clients. The client must specify which driver instance it wants to be a client to and IO mode in which the driver should be opened. The USB Device Driver while operating in USB device mode only supports non-blocking, read write and shared client access I/O intent mode. Trying to open the driver in any other mode will cause the open routine to fail.

The driver treats the first client that opens it, differently from the other clients. This first client must open a bidirectional pipe on Endpoint 0. It must then register a Setup Packet Buffer with the driver. The driver uses this buffer to store the 8-byte Setup command details received during the Setup stage of a Control transfer. While all clients can register a USB event callback function, only the first client receives the USB_DEVICE_SETUP_TOKEN and the USB_DEVICE_SETUP_HANDSHAKE event in its callback routine.

Typically (and it is recommended that) the first client to open the driver should be the USB Device Stack Device Layer Firmware. The Device Layer firmware is responsible for enumerating the USB device and responding to host requests on the Endpoint 0 message pipe. The first client should register a USB event callback function (by using the DRV_USB_Device_EventCallbackSet function) and Setup Packet Buffer (by using the DRV_USB_Device_SetupPacketBufferAssign function) with the Driver. Both of these are essential for operation of the driver. Registering a Setup Packet Buffer via the DRV_USB_Device_SetupPacketBufferAssign also enables the USB device (activates the USB module pull up resistors) for operation on the USB.

The following code shows an example of how a USB Device Stack Device Layer should open the driver and perform other functions.

```
/* In this example, the USB Device Layer (myUSBDevice) opens
 * the USB Driver. It is assumed that the Driver is already
 * initialized */

  myUSBDevice.usbDriverHandle = DRV_USB_Open(DRV_USB_INDEX_0,
              DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_SHARED  );

  if(DRV_HANDLE_INVALID == myUSBDevice.usbDriverHandle)
  {
      /* The open function failed. We can find out why */
      if(USB_CLIENT_NOTAVAIL == DRV_USB_Device_DriverClientErrorGet(DRV_USB_INDEX_0))
      {
          /* This means that there is no space for another client.
           * The driver configuration needs to be modified to support
           * more clients. This can be done by increasing the value
           * of DRV_USB_CLIENTS_NUMBER */

           while(1);
      }
  }

  /* Set the client event callback for the Device Layer.
   * The USBDeviceLayerEventHandler() function is the event
   * handler. When this event handler is invoked by the
   * driver, the driver returns back the second argument
   * specified in the following function (which in this case
   * is the device layer data structure). This allows the
   * application firmware to identify, as an example, the
   * device layer object associated with this callback
   */

  DRV_USB_Device_EventCallBackSet(myUSBDevice.usbDriverHandle, (void *)&myUSBDevice,
                                            USBDeviceLayerEventHandler);

  /* Device Layer register setup packet buffer. The Setup Packet Buffer
   * size is 8 bytes and is of the type uint8_t. Calling this function
   * enables the device and the device is ready to receive a Setup token
   * from the host. */

  DRV_USB_Device_SetupPacketBufferAssign(myUSBDevice.usbDriverHandle, setupPacketBuffer);

  /* Device Layer opens a endpoint 0 pipe. Refer to Setting
   * up a Pipe section for more details on how to set up the
```

```
     * pipe. This may be done in the reset event.*/

    myUSBDevice.ep0Pipe = DRV_USB_Device_PipeSetup(myUSBDevice.usbDriverHandle, 0, USB_EP_TX_RX,
                                64, 1, USB_CONTROL_PIPE);
```

A function driver may open the driver for use based on the configuration set by the host. Function drivers may or may not require USB Event notification and so may not need to register a USB Event Callback function. The user application can also open the driver. This allows the user application to track USB events and also communicate directly on the bus. The following code shows an example of how a function driver can open the USB Driver.

```
  /* This code shows an example of how the
   * a function driver, say CDC, could open the USB Driver.
   * Note that this should happen on a set configuration
   * event from the host. */

    cdc.usbDriverHandle = DRV_USB_Open(DRV_USB_INDEX_0,
                DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE  );

    if(DRV_HANDLE_INVALID == cdc.usbDriverHandle)
    {
        /* The open function failed. We can find out why */
        if(USB_CLIENT_NOTAVAIL == DRV_USB_Device_DriverClientErrorGet(DRV_USB_INDEX_0))
        {
            /* This means that there is no space for another client.
             * The driver configuration needs to be modified to support
             * more clients. This can be done by increasing the value
             * of DRV_USB_CLIENTS_NUMBER */ */

             while(1);
        }
    }

    /* Set the USB Event handler for CDC. Note that
     * this optional and should be done only if the
     * function driver requires USB Events notification. */

    DRV_USB_Device_EventCallBackSet(cdc.usbDriverHandle, (void *)&cdc,
                                            CDCDeviceUSBEventHandler);

    /* USB CDC Device Driver opens a TX and RX bulk pipe on
     * endpoint 1. */

    cdc.bulkRxPipe = DRV_USB_Device_PipeSetup(cdc.usbDriverHandle, 1,
                            USB_EP_RX, 64, 3, USB_BULK_PIPE);

    cdc.bulkTxPipe = DRV_USB_Device_PipeSetup(cdc.usbDriverHandle, 1,
                            USB_EP_TX, 64, 3, USB_BULK_PIPE);
```

Endpoint Pipes

A client must open a pipe to schedule transfers on the USB. This is done by using the DRV_USB_Device_PipeSetup function. From a USB perspective, an Endpoint 0 pipe is a message pipe (or a control pipe). The data flowing through this pipe has a structure which defined by the USB specification. Pipes on non-zero endpoints are stream pipes and need not follow a specific format. A Pipe connects a device's endpoint to a Host Driver software. Endpoint 0 pipe is always needed and must be created for the USB device to enumerate and therefore operate correctly. Non-zero endpoint pipes are created based on the configuration set by the Host.

Pipe Setup

The DRV_USB_Device_PipeSetup function creates a pipe. The following should be noted while using this function to create a pipe.

- The pipeDirection argument specifies the pipe direction. An endpoint 0 pipe is always bidirectional. So this parameter is ignored when the nEndpoint parameter is 0. All other pipes must be unidirectional. Specifying USB_EP_TX as the pipe direction sets up the pipe to send data from device to host (host Read transfers). Specifying USB_EP_RX as the pipe direction sets up the pipe to receive data from host (host Write transfers).
- The nBytesMaxSize parameter specifies the maximum size of the data packet in a transaction on that pipe. It should not exceed the maximum endpoint size communicated to the host during enumeration. Depending on the type of the USB transfer that the pipe will handle and the speed of the USB, the maximum size of the data packet may vary. If the size of the read transfer (device to host) scheduled on the pipe is greater than nBytesMaxSize, the driver will respond correctly multiple transactions that are issued by the host to complete the transfer.
- The nXferQueueSize parameter defines the size of the transfers that can be queued up this pipe. The minimum queue size should be 1. The control pipe (Endpoint 0 pipe) always has queue size of 1. Control transfers on the endpoint 0 pipe cannot be queued. Refer to the Transfer Request section for more information on how transfers are scheduled. Other non-zero endpoint pipes can have a queue size greater than or equal to '1'.
- An endpoint and direction of communication cannot be shared between two pipes. For example if a client opens a RX pipe on Endpoint 1, another RX pipe on Endpoint 1 cannot be opened.
- The pipeType parameter specified the type of transfers that this pipe will handle.

The client can check the return value of the DRV_USB_Device_PipeSetup for the result. If the pipe was not set up successfully, the function returns DRV_HANDLE_INVALID. The client can then use the DRV_USB_Device_ClientPipeErrorGet function to find out the reason for the failure.

The following code shows how endpoint 0 pipe and non-zero endpoint pipes are created.

```
/* This is an example of how an endpoint 0 pipe is created.
 * Note that this pipe is always bidirectional and the
 * queue size is always 1. The maximum data packet size in this
 * case is 64. The pipe type is set to handle control transfers */

myUSBDevice.ep0Pipe = DRV_USB_Device_PipeSetup(myUSBDevice.usbDriverHandle, 0, USB_EP_TX_RX,
                            64, 1, USB_CONTROL_PIPE);

if(DRV_HANDLE_INVALID == myUSBDevice.ep0Pipe)
{
     switch(DRV_USB_Device_ClientPipeErrorGet(myUSBDevice.usbDriverHandle))
     {
          case USB_PIPE_QUEUE_NOT_AVAILABLE:

              /* This means that queue size is small. Increase
               * the value of DRV_USB_XFERS_NUMBER. */

               break;

          case USB_PIPE_BAD_ENDPT:

              /* The specified endpoint is greater than the
               * the maximum end points that driver is configured
               * to handle. */

               break;
```

```
            case USB_PIPE_TAKEN:

                /* The specified endpoint and direction are already
                 * in use with another pipe. A non-zero endpoint
                 * can have only one pipe in a direction. Endpoint
                 * 0 can have one bidirectional pipe.*/

                 break;

            case USB_PIPE_NOTAVAIL:

                /* No pipes are available. Increase the number of pipes
                 * by increasing the value of DRV_USB_PIPES_NUMBER. */

                 break;

            case USB_PIPE_BAD_DIRECTION:

                /* Non zero endpoint cannot be bidirectional. */

                 break;
        }

    /* This is an example of how a client creates a non-zero endpoint pipe.
     * In this case a TX and a RX pipe are opened on endpoint 1. The
     * transfer schedule queue size is 4 and maximum data packet size
     * is 64. Note that the maximum data packet size should match endpoint
     * size specified in the endpoint configuration. */

    cdc.bulkRxPipe = DRV_USB_Device_PipeSetup(cdc.usbDriverHandle, 1, USB_EP_RX, 64, 3,
                                                   USB_BULK_PIPE);

    /* Optionally check for error here by using the
     * DRV_USB_Device_ClientPipeErrorGet() function */

    cdc.bulkTxPipe = DRV_USB_Device_PipeSetup(cdc.usbDriverHandle, 1, USB_EP_TX, 64, 3,
                                                   USB_BULK_PIPE);

    /* Optionally check for error here by using the
     * DRV_USB_Device_ClientPipeErrorGet() function */

    /* The following usage of DRV_USB_Device_PipeSetup() is INCORRECT and
     * will not work. The client must create separate TX and RX pipes for
     * non-zero endpoints. This function will return DRV_HANDLE_INVALID
     * and the DRV_USB_Device_ClientPipeErrorGet()function will return
     * USB_PIPE_BAD_DIRECTION. */

     someDevice.bulkPipe = DRV_USB_Device_PipeSetup(someDevice.usbDriverHandle, 1,
                                    USB_EP_TX_RX, 64, 3, USB_BULK_PIPE); // Will not work!
```

Pipe Status

The client can use the DRV_USB_Device_PipeStatusGet function to obtain the status of the pipe. The client can use this information to find if a pipe is ready for transfers or if the pipe can be or is stalled. The following code shows an example of the DRV_USB_Device_PipeStatusGet function along with all the possible return values.

```
/* This code example assumes the DRV_USB_DevicePipeSetup() function
 * was used to open a pipe */

DRV_USB_PIPE_HANDLE pipeHandle;
DRV_USB_PIPE_STATUS pipeStatus;

if(DRV_HANDLE_INVALID != pipeHandle)
{
    pipeStatus = DRV_USB_Device_PipeStatusGet(pipeHandle);
    switch (pipeStatus)
```

```
    {
        case USB_PIPE_CLOSED:
            /* The pipe was open and has been closed.
             * This pipe should not be used for transferring
             * data */
            break;
        case USB_PIPE_XFER_IN_PROGRESS:
            /* There is a transfer in progress on the USB
             * through this pipe.
             */
            break;
        case USB_PIPE_READY:
            /* Pipe is ready for transferring data. Also
             * indicates that there is no transfer in progress
             * on the USB through this pipe
             */
            break;
        case USB_PIPE_STALLED:
            /* Pipe is stalled. No transfers will take place.
             */
            break;
    }
}
```

Pipe Close

An open pipe can be closed by using the DRV_USB_Device_PipeRelease function. Calling this function performs the following:

- Aborts all transfers that are in queue. If a transfer is in progress on the USB, it is canceled.
- Deallocates its transfer queue and returns it to the driver
- Deallocates the pipe and returns its to the driver

The DRV_USB_Device_PipeRelease function may be called when the device stack detects detach from the host, or when a USB reset event had been detected. Because this function aborts transfers that are in progress on the USB, it is possible that calling this function while a transfer is in progress can cause the USB protocol to be violated. To avoid this, the client can use the DRV_USB_Device_PipeStatusGet function to get the status of the pipe before closing it. In case of a USB reset event, the client can close the pipe without checking pipe status as the driver itself will cancel all transfer that are in progress.

The following code shows example usage of the DRV_USB_Device_PipeRelease function.

```
/* This code assumes that this is not a reset
 * event. In such a case, it is recommended that any
 * transfers through the pipe be allowed to complete.
 */

while(DRV_USB_Device_PipeStatusGet(pipeHandle) == USB_PIPE_XFER_IN_PROGRESS);

/* Close the pipe */

DRV_USB_Device_PipeRelease(pipeHandle);
```

Pipe Transfer Queue

A pipe provides a queue for transfer requests. Clients can use the queue to reduce wait times and therefore increase data throughput through the pipe. If the pipe queue size is greater than one, the transfer requests on the pipe will be added to the queue if a transfer is already in progress on the pipe. The size of queue is specified via the nXferQueueSize parameter in the DRV_USB_Device_PipeSetup function. The queue contains transfer objects that hold information about the requested transfers. Multiple driver instance share one aggregate queue. The size of aggregate queue is defined by the DRV_USB_XFERS_NUMBER macro in the driver configuration. Pipe specific queues are formed from this aggregate queue. Figure 2 shows how this works.

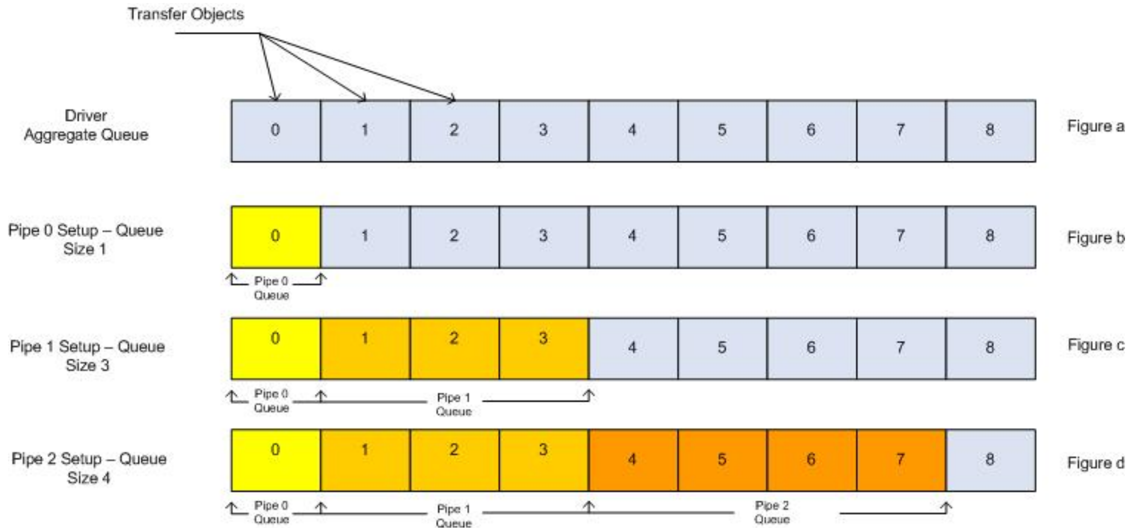


Figure 2: Pipe Transfer Queues.

Figure 2a shows the aggregate driver queue. In this case, all the transfer objects in the queue are available for use. In Figure 2b, the client creates a pipe (pipe 0) with a queue size of 1. This results in the first transfer object on the queue forming a single element queue for pipe 0. In Figure 2c, the client creates a pipe (pipe 1) with a queue size of 3. This results in the next three transfer objects on the queue forming a 3 element queue for pipe 1. In Figure 2d, the client creates a pipe (pipe 2) with a queue size of 4. This results in the next four transfer objects on the queue forming a 4 element queue for pipe 2. At this point, if a client tries to create another pipe with queue size of say 4, the pipe will not be created as there isn't enough space available in the queue.

The size of the aggregate queue should be decided in such a way as to meet the application's dynamic requirements while taking the memory availability into consideration.

Pipe Stall

The client can stall a pipe in response to USB events. When a pipe is stalled, it returns a STALL PID on the USB. The USB device firmware may stall endpoint 0 pipe when control requests from the host fail or are not supported. The driver automatically clears an endpoint 0 pipe stall when the host sends a SETUP packet. A non-zero endpoint pipe may be stalled when the endpoint's Halt feature is set. This is a functional stall. In this case the endpoint cannot send or receive data. Endpoint 0 pipe may not need to support functional stalls (although they can but this is rarely done). A pipe is stalled using the DRV_USB_Device_PipeStall function and a pipe stall is cleared using the DRV_USB_Device_PipeStallClear function.

There are three execution threads where the driver client can stall a pipe.

A pipe can be stalled in the client event handler:

The client can stall the pipe after receiving the USB_DEVICE_SETUP_TOKEN event. This is relevant to control transfer on Endpoint 0 pipe. The firmware can stall the pipe if it does not support the command received in the Setup packet. It is recommended that both direction of the Endpoint 0 pipe be stalled (this is possible because Endpoint 0 pipe is always bidirectional). Figure 3 shows the transaction related to this case.

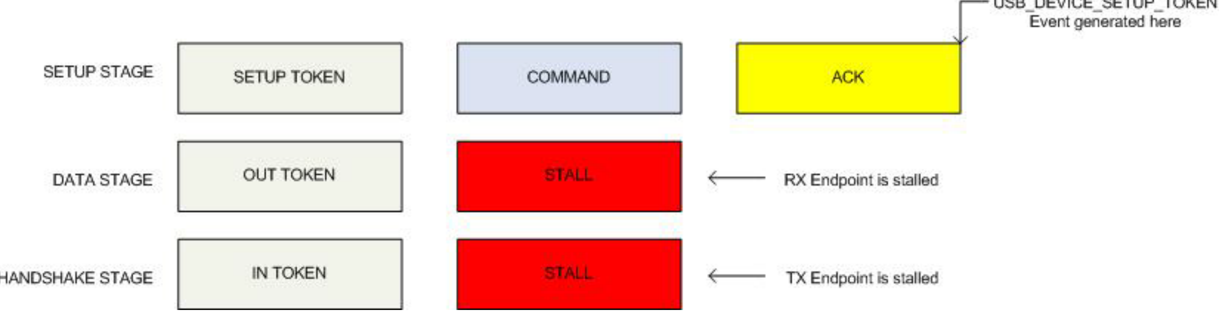


Figure 3: Stalling Endpoint 0 pipe on a Control Read Transfer (Pipe is stalled after

USB_DEVICE_SETUP_TOKEN event)

The stall on the receive direction of the pipe will get cleared when the host issues a Setup token. The stall on the TX direction of the pipe needs to cleared by using the DRV_USB_Device_PipeStallClear function. For an Endpoint 0 pipe, it is recommend that the stall on both directions be cleared. Note that is also possible to stall non-zero pipes in the USB_DEVICE_SETUP_TOKEN event (see the following code example).

```
void USBEventHandler (void * referenceHandle,
                        DRV_USB_EVENT  eventType, DRV_USB_EVENT_DATA * eventData )
{
    switch ( eventType )
    {

        /* There could be other events here as well. Only
         * the USB_DEVICE_SETUP_TOKEN is shown in this example
         * for the sake of clarity */

        case USB_DEVICE_SETUP_TOKEN:
            // Setup token received from host

            if(ProcessSetupPacket(eventData->setupEventData.pSetupPktBuffer) == false)
            {
                /* This means the command received in the setup packet is
                 * not supported. The endpoint should be stalled. While
                 * stalling endpoint 0, both directions should be stalled.
                 * This ensures that both data stage and handshake stage
                 * of the control transfer get stalled. */

                 DRV_USB_Device_PipeStall(pipe0, USB_EP_TX_RX);
            }
            else
            {
                /* Setup packet command can be processed */
                if(DRV_USB_Device_PipeStatusGet(pipe0) == USB_PIPE_STALLED)
                {
                    /* Clear the pipe stall */
                    DRV_USB_Device_PipeStallClear(pipe0, USB_EP_TX_RX);
                }
            }
            break;
    }
}
```

A pipe can be stalled in the transfer complete event handler:

This is more relevant to Control Write transfers. In the case of Control Write transfers, the transfer complete event handler is called after the data stage of the control transfer. If the firmware does not support the contents of the data stage, it can stall the pipe. This will result in the handshake stage of the control write transfer getting stalled. Figure 4 shows the transactions related to this case.

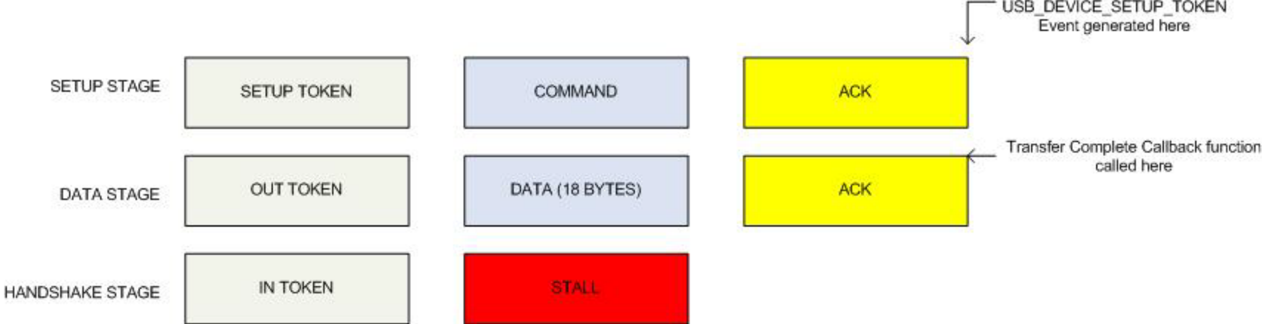


Figure 4: Stalling Endpoint 0 pipe on a Control Write Transfer (Pipe is stalled in the Transfer Complete Callback Function)

The following code shows an example of this:

```
void USBDeviceLayerXferCallback( void *  referenceData,
                           DRV_USB_PIPE_HANDLE hPipe,
                           DRV_USB_XFER_HANDLE hTransfer,
                           unsigned short int  transferByteCount  ,
```

```
                            DRV_USB_DEVICE_XFER_STATUS  statusTransfer )
{
    /* Note that the transfer call contains the pipe
     * handle. This pipe handle can be used to
     * stall the pipe */

    DRV_USB_Device_PipeStall(hPipe, USB_EP_TX_RX);
}
```

A pipe can be stalled asynchronous of driver events:

The client can stall a pipe based on device firmware logic. This is more applicable to non-zero endpoint pipes. Since these pipes are unidirectional, the client must specify the stall direction (USB_EP_TX or USB_EP_RX) while calling the DRV_USB_Device_PipeStall function. Using the USB_EP_TX parameter will stall read transfers (IN tokens). Using the USB_EP_RX parameter will stall write transfers (OUT tokens).

Event Handling

Any client that opens the driver can register to receive USB events. These USB events can be used by the clients to achieve the desired functionality of the USB device. The first client to open the driver (typically the USB Stack device layer firmware) will receive additional events. The client uses the DRV_USB_Device_EventCallBackSet function to set the client event callback function. The following code shows a client event handler function.

```
/* This is the client event handler */
void ClientEventHandler (void * referenceHandle,
                                DRV_USB_EVENT  eventType,
                                DRV_USB_EVENT_DATA * eventData )
{
    switch ( eventType )
    {
        case USB_ERROR:
            // Bus error occurred and was reported in error interrupts
            break;
        case USB_DEVICE_RESET:
            // Host has commanded a device reset
            break;
        case USB_DEVICE_RESUME:
            // Resume detected while USB in suspend mode
            break;
        case USB_IDLE_DETECT:
            // Idle detected
            break;
        case USB_STALL:
            // Stall handshake has occurred
            break;
        case USB_DEVICE_SETUP_TOKEN:
            // A Setup token was received
            break;
        case USB_DEVICE_SETUP_HANDSHAKE:
            // Handshake stage of the control
            // transfer is completed.
            break;
            // Device received a Start of Frame
        case USB_DEVICE_GOT_SOF
            break;
        default:
            break;
    }
}

/* The client register the event handler
 * after it has opened the driver. referenceData
 * will be passed back to the event callback function
 * whenever it is invoked. */

DRV_USB_Device_EventCallBackSet(clientHandle, referenceData, ClientEventHandler);
```

The eventData argument in the event callback function should be interpreted according to the event. The following table shows all the possible events and the value of eventData for each event.

Event Type	Event Details	Interpretation of eventData
USB_ERROR	This event occurs when the USB module has detected an error on the bus	eventData.usbErrorData.errorType contains a bitmap of errors (defined by USB_ERROR_TYPE) that have occurred.
USB_DEVICE_RESET	This event occurs when the USB module has detected a USB Reset	NULL
USB_DEVICE_RESUME	This event occurs when the USB module has detected Resume Signalling	NULL

USB_IDLE_DETECT	This event occurs when the USB module has not detected any bus activity	NULL
USB_STALL	This event occurs when a stalled endpoint has received a token	eventData.stalledEndpoint.endpoint contains the stalled endpoint that received a token.
USB_DEVICE_SETUP_TOKEN	This event occurs when endpoint 0 has received a SETUP token	eventData.setupEventData.pSetupPktBuffer points to a 8 byte array that contains the setup command that was received.
USB_DEVICE_SETUP_HANDSHAKE	This event occurs when the handshake stage of a Control Write Transfer has completed.	NULL
USB_DEVICE_GOT_SOF	This event occurs when the USB module has received a start of frame.	NULL

The following code shows the different USB errors that can occur

```
USB_ERROR_TYPE error = eventData.usbErrorData.errorType;

/* These are the possible errors that can occur.
 * Refer to the device specific data sheet for
 * more details on these errors. */

if (errorType & USB_ERR_INT_PID_CHECK_FAILURE)
{
    /* Invalid or unsupported PID received */
}
if ( errorType & USB_ERR_INT_BAD_CRC5)
{
    /* A CRC 5 error was encountered */
}
if ( errorType & USB_ERR_INT_BAD_CRC16)
{
    /* A CRC 16 error was encountered */
}
if ( errorType & USB_ERR_INT_BAD_DATA_FIELD_SIZE)
{
    /* Unsupported data field size was encountered */
}

if ( errorType & USB_ERR_INT_BUS_TURNAROUND_TIMEOUT)
{
    /* A bus turn around timeout was encountered */
}

if ( errorType & USB_ERR_INT_DMA_ERROR)
{
    /* A USB DMA error was encountered */
}

if ( errorType &= USB_ERR_INT_BUS_MATRIX_ERROR)
{
    /* A Bus matrix error was encountered */
}

if ( errorType & USB_ERR_INT_BIT_STUFF_ERROR)
{
    /* A bit stuffing error was encountered */
}
```

Transfer Requests

A USB Driver (operating in Device mode) client can receive and transmit data from the USB Host after creating a pipe. Data is transferred by using the DRV_USB_Device_TransferRequest function. Four types of USB transfers are supported: Control, Bulk, Interrupt and Isochronous. The choice of transfer type depends on the nature of desired communication and the target functionality of the USB device. Requesting a transfer will cause the USB driver to schedule the transfer. The actual data communication takes place when the host has requested for the data. In cases where a transfer is already in progress and space is available on the pipe transfer queue, the transfer request is added to the queue and processed in turn. Transfers that are not in progress (pending in the queue) can be aborted by using the DRV_USB_Device_TransferAbort function.

Depending on the size of the endpoint and the amount of data to be transferred, the host may complete the transfer through multiple transactions. This process is handled by the driver and is transparent to the client. The client can register a transfer complete event callback function for each transfer request. This function is called when the transfer is terminated, either because its completed or aborted. Optionally the client can call the DRV_USB_Device_TransferStatusGet function to poll the status of the transfer. The transfer complete event callback function need not be registered in this case.

The DRV_USB_Device_TransferRequest function also accepts flags that affect the processing of the transfer. The flag options are transfer type specific.

The following sections provide more details on how the four different transfer types can be requested.

Control Transfers

The USB Driver supports Control Read and Write Transfers on a endpoint 0 pipe only. A control transfer (either read or write) occurs in stages:

- The SETUP stage provides a setup command, which among other things, also defines if the transfer is a Control Read (data moves from device to host) or a Control Write transfer (data moves from host to device)
- The DATA stage transports data related to the setup command in the SETUP stage. Depending on the size of the endpoint, the data stage could comprise of multiple IN or OUT transactions. A data stage may or may not be present in Control Write transfers.
- The HANDSHAKE stage completes the Control transfer

Control Transfers should only be requested in response to a setup packet from the host. The USB Driver notifies its first client (via the client event callback function) when a SETUP packet has been received. Only the first client receives this, USB_DEVICE_SETUP_TOKEN, event. The driver generates USB_DEVICE_SETUP_HANDSHAKE client event at the completion of the handshake stage of the control write transfer. Refer to Event Handling section on more details on handling events.

The client parses the 8-byte setup command available from the USB_DEVICE_SETUP_TOKEN, event. It then has the following options:

- The client can request a control read transfer
- The client can request a control write transfer
- The client may not do anything and return from the event handler. In this case, the control transfer is treated as having no data stage and the transfer moves to the handshake stage. Note that a Control Read transfer must always have a data stage.
- The client can stall the pipe

These cases are described here in detail.

<u>Requesting a Control Read Transfer:</u>

The following code shows an example of how a Control Read Transfer can be requested. In this transfer, data moves from the device to the host.

```
/* This is an example of a Control Read Transfer. A control read
 * or Write transfer should be requested in response to a
 * USB_DEVICE_SETUP_TOKEN event. The transfer request function is
 * called in the USB event handler. In this example, the amount of
 * data to be transferred is 18 bytes. The transfer is requested
```

```
 * over endpoint 0 pipe. The USBDeviceLayerXferCallback() function
 * will be called when the transfer is terminated. The myUSBDevice
 * object will be passed back to the USBDeviceLayerXferCallback()
 * function. The USB_XFER_REGULAR indicates to the driver that this
 * transfer does not need a data stage ZLP */

DRV_USB_XFER_HANDLE xferHandle;

void USBEventHandler (void * referenceHandle,
                        DRV_USB_EVENT  eventType, DRV_USB_EVENT_DATA * eventData )
{
    switch ( eventType )
    {

        /* There could be other events here as well. Only
         * the USB_DEVICE_SETUP_TOKEN is shown in this example
         * for the sake of clarity */

        case USB_DEVICE_SETUP_TOKEN:
            // Setup token received from host

            ProcessSetupPacket(eventData->setupEventData.pSetupPktBuffer);
            xferHandle = DRV_USB_Device_TransferRequest(myUSBDevice.ep0Pipe,
                                    USB_XFER_CONTROL_READ, deviceDescriptor,18,
                                    USB_XFER_REGULAR, myUSBDevice,
                                    USBDeviceLayerXferCallback);

            if(DRV_HANDLE_INVALID == xferHandle)
            {
                /* If the transfer request failed, the
                 * client can find out why. */

                switch(DRV_USB_Device_PipeXferErrorGet(myUSBDevice.ep0Pipe))
                {
                    case USB_XFER_WRONG_PIPE:
                       /* This means that either the transfer type or
                        * does not match the pipe type or direction */
                        break;
                    case USB_XFER_QUEUE_FULL:
                       /* This means that the transfer queue on this
                        * pipe is full. Either increase the queue size
                        * by increasing the value of DRV_USB_XFERS_NUMBER
                        * or wait for a slot on the queue to be available. */
                        break;
                    default:
                        break;
                }
            }
      }
}
```

In a case where the data to be transferred is more than the endpoint size reported to the host (also specified at the time when the control endpoint pipe is set up), the host will break up the data stage of the transfer into multiple transactions. This process is transparent to the client. If at any point a SETUP packet is received while the transfer was in progress, the transfer is aborted and client receives USB_DEVICE_SETUP_TOKEN event. If a transfer complete callback function was registered in the transfer request function, this callback function is called with transfer status as USB_XFER_ABORT. Figure 5 shows the generation of different events during the Control Read Transaction

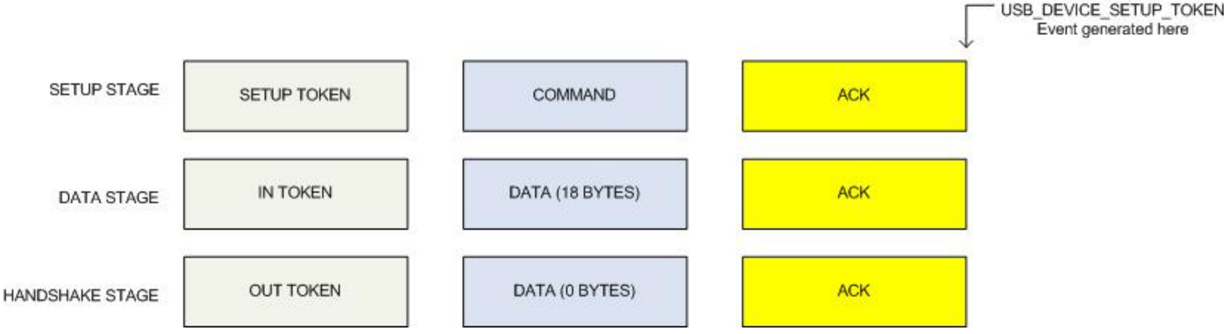


Figure 5: Control Read Transfer. Data size is 18 bytes (Maximum Endpoint size is 64 bytes (> 18 bytes))

A Control read transfer must have a data stage. In cases of control read transfers where the amount of data to be transferred is less than that requested by the host, the USB_XFER_SEND_ZLP flag must be specified. The driver will then check if the transfer size is an exact multiple of the endpoint size and then send a data stage zero length packet (ZLP). This indicates to the host that the device does not have any more data. There are two cases where specifying the USB_XFER_SEND_ZLP flag does not cause the driver to send a ZLP:

- When the size of the transfer is less than requested size but is not an exact multiple of the maximum endpoint size
- The size of the transfer is less than maximum endpoint size

The following code shows an example of a transfer request which sends a ZLP to the host.

```
/* This is an example of a Control Read Transfer with the ZLP
 * option. Endpoint 0 maximum size is 8 bytes. The host has
 * requested 32 bytes but the device has only 24 bytes to send.
 * Specifying the USB_XFER_SEND_ZLP flag will cause the driver
 * to send a ZLP to the host in the fourth data transaction.
 */

 DRV_USB_XFER_HANDLE xferHandle;

 xferHandle = DRV_USB_Device_TransferRequest(myUSBDevice.ep0Pipe, USB_XFER_CONTROL_READ,
                             deviceDescriptor,24,USB_XFER_SEND_ZLP,
                             myUSBDevice, USBDeviceLayerXferCallback);
```

Figure 6 shows the generation of different events and the additional ZLP sent in data stage in a multi transaction Control Read Transfer.

Figure 6: Multi Transaction Control Read Transfer with ZLP (Data size is 24 bytes and maximum endpoint size is 8 bytes)

Requesting a Control Write Transfer:

The following code shows an example of how a Control Write Transfer can be requested. In this transfer, data moves

from the host to device. The following code example shows how a control write transfer is requested. Figure 7 shows the generation of different events related to the control write transfer.

```
/* This is an example of a Control Write Transfer. A control read
 * or Write transfer should be requested in response to a
 * USB_DEVICE_SETUP_TOKEN event. The transfer request function is
 * called in the USB event handler. In this example, the amount of
 * data to be transferred is 32 bytes. The transfer is requested
 * over endpoint 0 pipe. The USBDeviceLayerXferCallback() function
 * will be called when the transfer is terminated. The myUSBDevice
 * object will be passed back to the USBDeviceLayerXferCallback()
 * function. The Control Write Transfer only supports the
 * USB_XFER_REGULAR flag.  */

DRV_USB_XFER_HANDLE xferHandle;

void USBDeviceLayerXferCallback( void *  referenceData,
                           DRV_USB_PIPE_HANDLE hPipe,
                           DRV_USB_XFER_HANDLE hTransfer,
                           unsigned short int  transferByteCount  ,
                           DRV_USB_DEVICE_XFER_STATUS  statusTransfer )
{
    /* This function will be called when the transfer is complete.
     * In this case, the referenceData argument will point to the
     * myUSBDevice data object that was specified in the Transfer
     * request function. */
}


void USBEventHandler (void * referenceHandle,
                        DRV_USB_EVENT  eventType, DRV_USB_EVENT_DATA * eventData )
{
    switch ( eventType )
    {

        /* There could be other events here as well. Only
         * the USB_DEVICE_SETUP_TOKEN is shown in this example
         * for the sake of clarity */

        case USB_DEVICE_SETUP_TOKEN:
            // Setup token received from host

            ProcessSetupPacket(eventData->setupEventData.pSetupPktBuffer);
            xferHandle = DRV_USB_Device_TransferRequest(myUSBDevice.ep0Pipe,
                                  USB_XFER_CONTROL_WRITE, buffer,32,
                                  USB_XFER_REGULAR, myUSBDevice,
                                  USBDeviceLayerXferCallback);

            if(DRV_HANDLE_INVALID == xferHandle)
            {
               /* The DRV_USB_Device_PipeXferErrorGet() can be used to
                * find the error cause */

            }
      }
}
```

In addition to the USB_DEVICE_SETUP_TOKEN event, the driver generates the USB_DEVICE_SETUP_HANDSHAKE event when the Control Write Handshake stage is complete. This event is generated even when the Control Write transfer does not have a data stage.

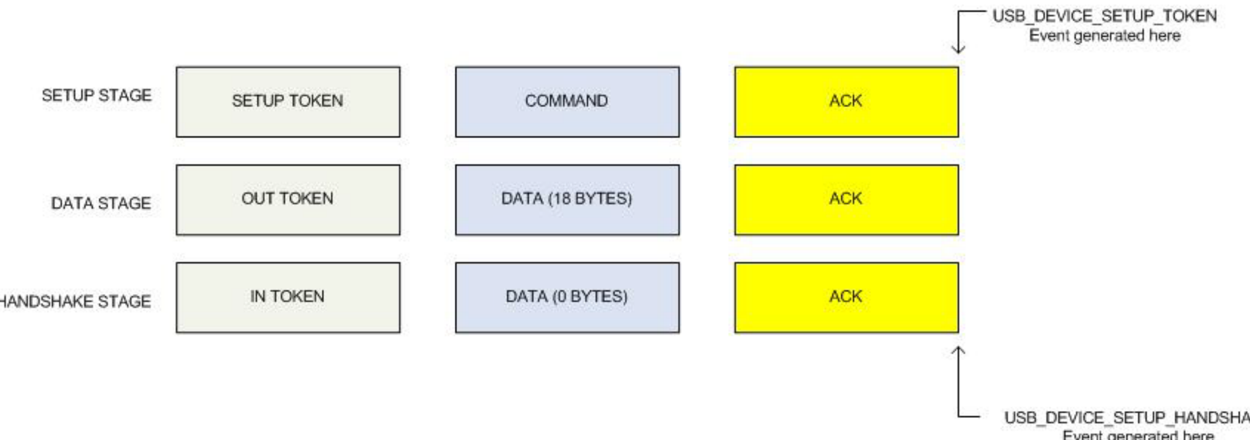


Figure 7: Example of Control Write Transfer (Data size is 18 bytes and maximum endpoint size is 64 bytes (> 18 bytes))

Control Write Transfers with Zero Data Stage:

A control write transfer can have zero data stages. One example of this is the SET ADDRESS command that the USB host send to the device during enumeration. In cases where no data stages are needed, the client should not request any transfer and return from the USB event handler. This will cause the driver to move the control transfer to the handshake stage. The USB_DEVICE_SETUP_HANDSHAKE function can be used to track when the handshake stage of the transfer completes. In case of the SET ADDRESS command, the device layer firmware can change the device USB address in response to this event. The following code example shows an example of how this is done.

```
/* This is an example of zero data stage Control Write transfer.
 * This is done by not issuing a transfer request after the
 * USB_DEVICE_SETUP_TOKEN event is received.  */

DRV_USB_XFER_HANDLE xferHandle;

void USBEventHandler (void * referenceHandle,
                        DRV_USB_EVENT  eventType, DRV_USB_EVENT_DATA * eventData )
{
    switch ( eventType )
    {

        /* There could be other events here as well. Only
         * the USB_DEVICE_SETUP_TOKEN is shown in this example
         * for the sake of clarity */

        case USB_DEVICE_SETUP_TOKEN:
            /* Setup token received from host. In this case we assume
             * that this is the SET ADDRESS command. This command does
             * not have a data stage. So the client does not do a
             * transfer request*/

            ProcessSetupPacket(eventData->setupEventData.pSetupPktBuffer);
            break;
        case USB_DEVICE_SETUP_HANDSHAKE:
            /* The last control write transfer request is complete.
             * In this example, the client can change the device USB address
             * now. */
            DRV_USB_Device_AddressSet(deviceLayer, address);
            break;
        default:
            break;
      }
}
```

Figure 8 shows the generation of different events related to the transfer.

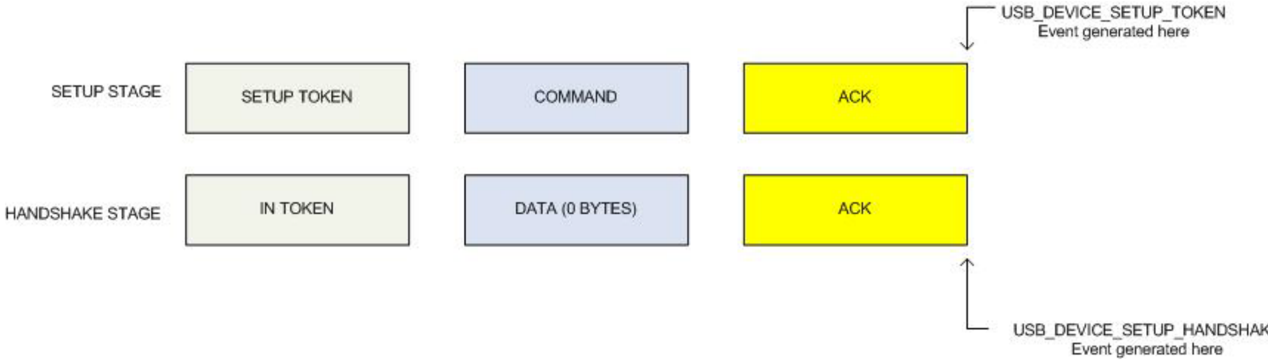


Figure 8: Example of Zero data stage Control Write Transfer

Stalling the Pipe:

Refer to Pipe Stall section for information on how a pipe can be stalled.

Bulk and Interrupt Transfers

The driver can process Bulk and Interrupt transfers. On the USB, both Bulk and Interrupt transfers have the same packet structure. The following discussion therefore applies to both Bulk and Interrupt transfers. A client should use the DRV_USB_Device_TransferRequest function to schedule Bulk and Interrupt transfers. The following sections describe Bulk/Interrupt Read Transfers and Bulk/Interrupt Write Transfers.

Bulk/Interrupt Read Transfers:

In this transfer, data moves from device to host. The client can request a bulk/interrupt read transfer with a byte size greater than the maximum endpoint size. In this case the driver will complete the transfer and invoke the transfer complete callback function if provided. If the USB_XFER_SEND_ZLP flag is specified, the driver sends a ZLP after the last transaction if the transfer size was an exact multiple of the endpoint size.

The following code shows an example of a Bulk/Interrupt read transfer can be requested

```
/* This code shows an example of how a Bulk/Interrupt
 * read transfer can be requested */

DRV_USB_XFER_HANDLE transferHandle;

transferHandle =  DRV_USB_Device_TransferRequest(pipe1, USB_XFER_BULK_READ,
                    data,120,USB_XFER_REGULAR, someReferenceData ,XferCompleteCallback);

/* This transfer directs the driver to send a ZLP
 * after the last transaction */

transferHandle =  DRV_USB_Device_TransferRequest(pipe1, USB_XFER_BULK_READ,
                    data,128,USB_XFER_SEND_ZLP, someReferenceData ,XferCompleteCallback);
```

Figure 9 shows a regular bulk/interrupt read transfer. Figure 10 shows a bulk/interrupt read transfer with a ZLP.

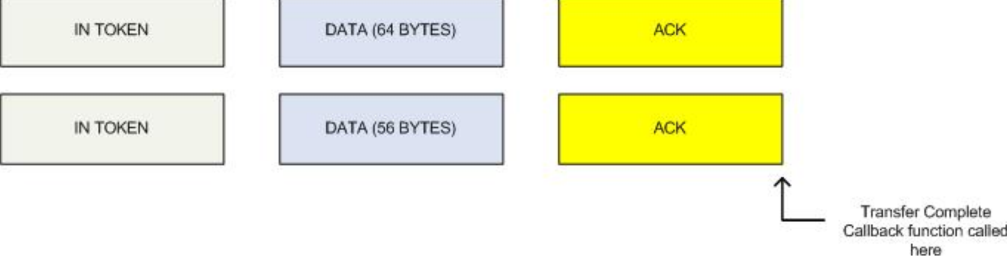


Figure 9: Example of Bulk/Interrupt Read Transfer (Requested transfer size is 120 bytes - Maximum endpoint size is 64)

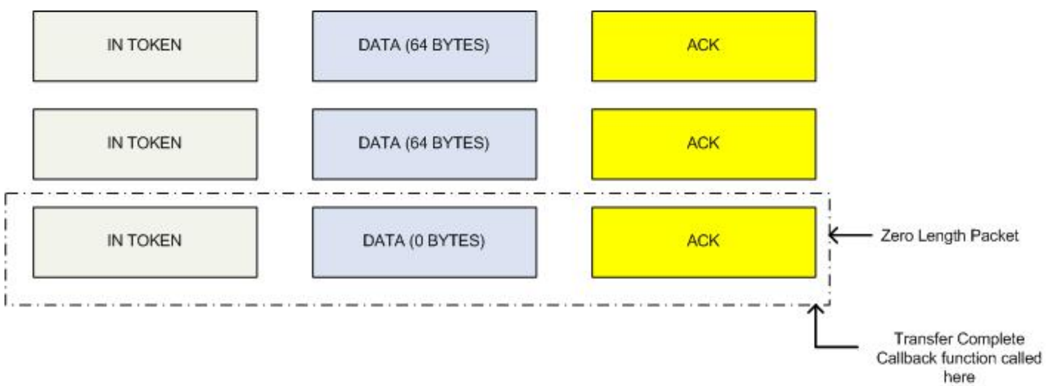


Figure 10: Example of Bulk/Interrupt Read Transfer with ZLP (Requested transfer size is 128 bytes - Maximum endpoint size is 64)

Bulk/Interrupt Write Transfers:

In this transfer, data moves from host to device. These transfers only support USB_XFER_REGULAR flag. The client can request a bulk/interrupt write transfer with a byte size greater than the maximum endpoint size. In this case however, the driver will invoke the transfer complete callback function when a transaction completes. The callback thus gets invoked whenever the host sends any amount of data. Since the amount of data that the host sends in one transaction will not exceed the maximum endpoint size, specifying a transfer size greater than the maximum endpoint size will have no effect. The client must schedule another transfer to receive more data from the host. The following code shows an example of bulk/interrupt write transfer.

```
/* This code shows an example of how a Bulk/Interrupt
 * write transfer can be requested. Note the data size is
 * equal the size of the endpoint. The client has queued up
 * multiple transfers.*/

DRV_USB_XFER_HANDLE transferHandle1;
DRV_USB_XFER_HANDLE transferHandle1;

transferHandle1 =  DRV_USB_Device_TransferRequest(pipe2, USB_XFER_BULK_WRITE,
                    data,64,USB_XFER_REGULAR, someReferenceData ,XferCompleteCallback);
transferHandle1 =  DRV_USB_Device_TransferRequest(pipe2, USB_XFER_BULK_WRITE,
                    data,64,USB_XFER_REGULAR, someReferenceData ,XferCompleteCallback);
```

Figure 11 shows a bulk/interrupt write transfer with related events

Figure 11: Example of Bulk/Interrupt Write Transfer (Client has scheduled two transfers - Maximum endpoint size 64)

Isochronous Transfers

A client can request Isochronous transfers by using the DRV_USB_Device_TransferRequest function. On the USB, Isochronous transfers are characterized by the absence of the handshake stage. The driver supports USB_XFER_REGULAR and USB_XFER_PERSIST flags for Isochronous transfers. While using the USB_XFER_REGULAR flag, the transfer request behavior is similar to Bulk and Interrupt Read/Write Transfer Requests. In that, a new transfer will require the client to call the DRV_USB_Device_TransferRequest function.

By using the USB_XFER_PERSIST flag, the driver will not discard the transfer after the transfer is complete. Instead, after the transfer is complete, the transfer is rescheduled in the queue. This way the client does not have to schedule

another transfer. If the client has scheduled two Isochronous transfers with the USB_XFER_PERSIST flag, the driver will ping pong between these transfers. The transfer complete callback function is called when the transfer is completed.

The following code shows an example of setting up two Isochronous read transfers to create a ping pong type of transfer system.

```
DRV_USB_XFER_HANDLE pingTransferHandle;
DRV_USB_XFER_HANDLE pongTransferHandle;

/* Request two transfers. Note how the USB_XFER_PERSIST flag
 * is specified. Note also how the pingData and pongData buffer
 * pointers are specified as reference data. This will make
 * processing in the transfer complete callback easier.. */

pingTransferHandle =  DRV_USB_Device_TransferRequest(pipe0, USB_XFER_CONTROL_READ,
                    pingData,64,USB_XFER_PERSIST, pingData ,PingXferCompleteCallback);

pongTransferHandle =  DRV_USB_Device_TransferRequest(pipe0, USB_XFER_CONTROL_READ,
                    pongData,40,USB_XFER_PERSIST, pongData ,PongXferCompleteCallback);

/* This is the Ping transfer Complete callback */
void PingCompleteCallback( void *  referenceData,
                           DRV_USB_PIPE_HANDLE hPipe,
                           DRV_USB_XFER_HANDLE hTransfer,
                           unsigned short int  transferByteCount  ,
                           DRV_USB_DEVICE_XFER_STATUS  statusTransfer )
{
    /* Note that we don't schedule one more request here.
     * We just add data to the buffer and return. This
     * same transfer will be processed again. The referenceData
     * points to the ping data buffer (as set in the transfer request
     * function. */

    void * data = referenceData;
    UpdateBuffer(data);
}

/* This is the Ping transfer Complete callback */
void PongCompleteCallback( void *  referenceData,
                           DRV_USB_PIPE_HANDLE hPipe,
                           DRV_USB_XFER_HANDLE hTransfer,
                           unsigned short int  transferByteCount  ,
                           DRV_USB_DEVICE_XFER_STATUS  statusTransfer )
{
    /* Note that we don't schedule one more request here.
     * We just add data to the buffer and return. This
     * same transfer will be processed again. The referenceData
     * points to the pong data buffer (as set in the transfer request
     * function. */

    void * data = referenceData;
    UpdateBuffer(data);
}
```

Transfer Status

The driver offers to different interface to track the status of a transfer, a polling interface and a callback interface. Both these interfaces can be invoked by the application. The two interfaces are discussed here:

Polling interface:

The client can access the polling interface by using the DRV_USB_Device_XferStatusGet function. This interface does not provide information when a transfer is aborted. The following code shows how the DRV_USB_Device_XferStatusGet function is used.

```
/* This code shows the different
 * values that the DRV_USB_Device_XferStatusGet()
```

```
 * function. This function implements the polling
 * interface for the transfer status.
 */

DRV_USB_DEVICE_XFER_STATUS transferStatus;

transferStatus = DRV_USB_Device_XferStatusGet(transferHandle);

switch(transferStatus)
{
    case USB_XFER_PENDING:
        /* This means the transfer is still in the transfer
         * queue. This transfer can be aborted.*/
    case USB_XFER_IN_PROGRESS:
        /* This means the transfers is currently in progress
         * on the bus. This transfer cannot be aborted. */
    case USB_XFER_COMPLETED:
        /* This means the transfer is complete */
    default:
        break;
}
```

Callback interface:

The callback interface to track the status of the transfer is implemented by using Transfer Complete Callback function. The client should specify the Transfer Complete Callback function when calling DRV_USB_Device_TransferRequest function. The driver call this function when a transfer has terminated, either due to the transfer completing or the transfer aborting. The Transfer Complete Callback function should be of the type DRV_USB_XFER_COMPLETE_CALLBACK. The driver returns the following information while calling the Transfer Complete Callback function.

- Handle to the pipe that contained this transfer
- Handle to the transfer
- Number of bytes read or written
- A reference to a data object that was specified in the DRV_USB_Device_TransferRequest for this transfer.
- Transfer termination status

The following code shows an example of how a Transfer Complete Callback function can be used.

```
/* This code shows an example of how to use the
 * transfer complete callback */

DRV_USB_XFER_HANDLE transferHandle;

/* Request a transfer. Note that the reference data pointer is set
 * to dataBuffer and the transfer complete callback function
 * XferCompleteCallback is specified. */

transferHandle =  DRV_USB_Device_TransferRequest(pipe0, USB_XFER_CONTROL_READ,
                    dataBuffer,18,false, dataBuffer ,XferCompleteCallback);

/* This is the transfer Complete callback */
void XferCompleteCallback( void *  referenceData,
                           DRV_USB_PIPE_HANDLE hPipe,
                           DRV_USB_XFER_HANDLE hTransfer,
                           unsigned short int  transferByteCount  ,
                           DRV_USB_DEVICE_XFER_STATUS  statusTransfer )
{
    /* The referenceData variable (the value of which
     * is specified in the transfer request function)
     * points to the data buffer associate with the
     * transfer. */

    void * dataBuffer = referenceData;

    if(transferStatus == USB_XFER_COMPLETED)
    {
        ProcessDataBuffer(dataBuffer, transferByteCount);
```

```
    }
    else if(transferStatus == USB_XFER_ABORTED)
    {
        /* This transfer was not successful. Take any
         * remedial action as needed. */
    }

}
```

Transfer Abort

A client can abort a USB transfer that it requested. This is achieved by using the DRV_USB_Device_TransferAbort function. A Control read or write transfer can automatically get aborted when the host issues a SETUP token out of turn. The DRV_USB_Device_TransferAbort will not abort a transfer if it is already in progress. However, closing the pipe will abort transfers that are in progress.

When a transfer is aborted, the Transfer Complete Callback function associated with the transfer is not invoked. If the Control Read or Write transfer was aborted due to a out of turn SETUP token from the host, Transfer Complete Callback function will be called and transfer status will be USB_XFER_ABORTED.

Configuring the Library

Macros

	Name	Description
	DRV_USB_INDEX	This is macro DRV_USB_INDEX.
	DRV_USB_INTERRUPT_SOURCE	This is macro DRV_USB_INTERRUPT_SOURCE.
	DRV_USB_PERIPHERAL_ID	This is macro DRV_USB_PERIPHERAL_ID.

Description

The configuration of the USB Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the USB Driver. Based on the selections made, the USB Driver may support the selected features. Some configuration settings will apply to all instances of the USB Driver while some configurations settings are applicable only if the driver is built for a static mode.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

Initialization Overrides

This section lists out the macros that need to be configured when building the static version of the driver. These values should be defined for correction operation of the driver, when the driver is built statically. Commenting the DRV_USB_INSTANCES_NUMBER macro will cause the driver to be built in a static mode.

Others

These macros affect all instances and all build types of the driver.

DRV_USB_INDEX Macro

File

drv_usb_config_template.h

C

```
#define DRV_USB_INDEX 0
```

Description

This is macro DRV_USB_INDEX.

DRV_USB_INTERRUPT_SOURCE Macro

File

drv_usb_config_template.h

C

```
#define DRV_USB_INTERRUPT_SOURCE INT_SOURCE_USB_1
```

Description

This is macro DRV_USB_INTERRUPT_SOURCE.

DRV_USB_PERIPHERAL_ID Macro

File

drv_usb_config_template.h

C

```
#define DRV_USB_PERIPHERAL_ID USB_ID_1
```

Description

This is macro DRV_USB_PERIPHERAL_ID.

Building the Library

This section lists the files that are available in the Graphics Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/usb`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_usb.h	This file should be included by any .c file that accesses the USB Driver API. This file contains the prototypes for the Device and Host mode driver API. Note that the USB driver API is the same for the USB controller on the PIC32MX and PIC32MZ driver.

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/usbfs/src/dynamic/drv_usb.c	This file should always be included in the project when using the PICMX USB Driver.
/usbhs/src/dynamic/drv_usb.c	This file should always be included in the project when using the PIC32MZ USB Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
usbfs/src/dynamic/drv_usb_device.c	This file should be included in the project if USB Device mode operation is required in the application, while using the PIC32MX USB Module.
usbhs/src/dynamic/drv_usb_device.c	This file should be included in the project if USB Device mode operation is required in the application, while using the PIC32MZ USB Module.
usbfs/src/dynamic/drv_usb_host.c	This file should be included in the project if USB Host mode operation is required in the application, while using the PIC32MX USB Module.
usbhs/src/dynamic/drv_usb_host.c	This file should be included in the project if USB Host mode operation is required in the application, while using the PIC32MZ USB Module.

Module Dependencies

The USB Driver for the PIC32MX microcontroller depends on the following modules:

- Interrupt System Service Library

The USB Driver for the PIC32MZ microcontroller depends on the following modules:

- Interrupt System Service Library
- Timer System Service Library
- Clock System Service Library

Library Interface

e) Data Types and Constants

	Name	Description
	DRV_USB_HOST_PIPE_HANDLE_INVALID	Defines the USB Driver Host Pipe Invalid handle
	DRV_USB_EVENT	Identifies the different events that the USB Driver provides.
	DRV_USB_EVENT_CALLBACK	Type of the USB event callback function
	DRV_USB_HOST_PIPE_HANDLE	Defines the USB Driver Host Pipe handle type
	DRV_USB_DEVICE_INTERFACE	Group of function pointers to the USB Driver Device Mode Client Functions.
	DRV_USB_HOST_INTERFACE	Group of function pointers to the USB Driver Host Mode Client Functions.
	DRV_USB_OPMODE	Possible operation modes of the USB Driver.
	DRV_USB_ROOT_HUB_INTERFACE	Group of function pointers to the USB Root Hub Functions.
	DRV_USB_DEVICE_ENDPOINT_ALL	This is macro DRV_USB_DEVICE_ENDPOINT_ALL.

Description

This section describes the APIs of the USB Driver. Refer to each section for a detailed description.

a) System Functions

b) Client Functions

c) Pipe Functions

d) Transfer Functions

e) Other Functions

e) Data Types and Constants

DRV_USB_HOST_PIPE_HANDLE_INVALID Macro

Defines the USB Driver Host Pipe Invalid handle

File

drv_usb.h

C

```
#define DRV_USB_HOST_PIPE_HANDLE_INVALID ((DRV_USB_HOST_PIPE_HANDLE)(-1))
```

Description

USB Driver Host Pipe Handle Invalid

Defines the USB Driver Host Pipe Invalid handle. The USB Driver should returns this value if the pipe could not be created.

Remarks

None.

DRV_USB_EVENT Enumeration

Identifies the different events that the USB Driver provides.

File

drv_usb.h

C

```
typedef enum {
  DRV_USB_EVENT_ERROR = 1,
  DRV_USB_EVENT_RESET_DETECT,
  DRV_USB_EVENT_RESUME_DETECT,
  DRV_USB_EVENT_IDLE_DETECT,
  DRV_USB_EVENT_STALL,
  DRV_USB_EVENT_SOF_DETECT,
  DRV_USB_EVENT_DEVICE_SESSION_VALID,
  DRV_USB_EVENT_DEVICE_SESSION_INVALID
} DRV_USB_EVENT;
```

Members

Members	Description
DRV_USB_EVENT_ERROR = 1	Bus error occurred and was reported
DRV_USB_EVENT_RESET_DETECT	Host has issued a device reset
DRV_USB_EVENT_RESUME_DETECT	Resume detected while USB in suspend mode
DRV_USB_EVENT_IDLE_DETECT	Idle detected
DRV_USB_EVENT_STALL	Stall handshake has occurred
DRV_USB_EVENT_SOF_DETECT	Either Device received SOF or SOF threshold was reached in the Host mode • operation
DRV_USB_EVENT_DEVICE_SESSION_VALID	Session valid
DRV_USB_EVENT_DEVICE_SESSION_INVALID	Session Invalid

Description

USB Driver Events Enumeration

Identifies the different events that the USB Driver provides. The USB driver should be able to provide these event to Device Layer.

Remarks

None.

DRV_USB_EVENT_CALLBACK Type

Type of the USB event callback function

File

drv_usb.h

C

```
typedef void (* DRV_USB_EVENT_CALLBACK)(DRV_HANDLE hClient, DRV_USB_EVENT eventType, void *
eventData);
```

Returns

None.

Description

Type of the USB Event Callback Function

Type of the USB event callback function. The client should register an event callback function of this type when it intends to receive events from the USB driver. The event callback function is registered using the DRV_USB_ClientEventCallBackSet() function.

Remarks

None.

Parameters

Parameters	Description
hClient	handle to driver client that registered this callback function
eventType	Event type
eventData	Event relevant data

DRV_USB_HOST_PIPE_HANDLE Type

Defines the USB Driver Host Pipe handle type

File

drv_usb.h

C

```
typedef uintptr_t DRV_USB_HOST_PIPE_HANDLE;
```

Description

USB Driver Host Pipe Handle

Defines the USB Driver Host Pipe handle type. The Host pipe handle returned by the USB driver should match this type.

Remarks

None.

DRV_USB_DEVICE_INTERFACE Structure

Group of function pointers to the USB Driver Device Mode Client Functions.

File

drv_usb.h

C

```
typedef struct {
  DRV_HANDLE (* open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
  void (* close)(DRV_HANDLE handle);
  void (* eventHandlerSet)(DRV_HANDLE handle, uintptr_t hReferenceData, DRV_USB_EVENT_CALLBACK
eventHandler);
  void (* deviceAddressSet)(DRV_HANDLE handle, uint8_t address);
  USB_SPEED (* deviceCurrentSpeedGet)(DRV_HANDLE handle);
  uint16_t (* deviceSOFNumberGet)(DRV_HANDLE handle);
  void (* deviceAttach)(DRV_HANDLE handle);
  void (* deviceDetach)(DRV_HANDLE handle);
  USB_ERROR (* deviceEndpointEnable)(DRV_HANDLE handle, USB_ENDPOINT endpoint,
USB_TRANSFER_TYPE transferType, uint16_t endpointSize);
  USB_ERROR (* deviceEndpointDisable)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
  USB_ERROR (* deviceEndpointStall)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
  USB_ERROR (* deviceEndpointStallClear)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
  bool (* deviceEndpointIsEnabled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
  bool (* deviceEndpointIsStalled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
  USB_ERROR (* deviceIRPSubmit)(DRV_HANDLE handle, USB_ENDPOINT endpoint, USB_DEVICE_IRP * irp);
  USB_ERROR (* deviceIRPCancelAll)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
  void (* deviceRemoteWakeupStart)(DRV_HANDLE handle);
  void (* deviceRemoteWakeupStop)(DRV_HANDLE handle);
  USB_ERROR (* deviceTestModeEnter)(DRV_HANDLE handle, USB_TEST_MODE_SELECTORS testMode);
} DRV_USB_DEVICE_INTERFACE;
```

Members

Members	Description
DRV_HANDLE (* open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);	This is a pointer to the driver open function
void (* close)(DRV_HANDLE handle);	This is pointer to the driver close function
void (* eventHandlerSet)(DRV_HANDLE handle, uintptr_t hReferenceData, DRV_USB_EVENT_CALLBACK eventHandler);	This is a pointer to the event call back set function
void (* deviceAddressSet)(DRV_HANDLE handle, uint8_t address);	This is a pointer to the device address set function
USB_SPEED (* deviceCurrentSpeedGet)(DRV_HANDLE handle);	This is a pointer to the device current speed get function
uint16_t (* deviceSOFNumberGet)(DRV_HANDLE handle);	This is a pointer to the SOF Number get function
void (* deviceAttach)(DRV_HANDLE handle);	This is a pointer to the device attach function
void (* deviceDetach)(DRV_HANDLE handle);	This is a pointer to the device detach function
USB_ERROR (* deviceEndpointEnable)(DRV_HANDLE handle, USB_ENDPOINT endpoint, USB_TRANSFER_TYPE transferType, uint16_t endpointSize);	This is a pointer to the device endpoint enable function

USB_ERROR (* deviceEndpointDisable)(DRV_HANDLE handle, USB_ENDPOINT endpoint);	This is a pointer to the device endpoint disable function
USB_ERROR (* deviceEndpointStall)(DRV_HANDLE handle, USB_ENDPOINT endpoint);	This is a pointer to the device endpoint stall function
USB_ERROR (* deviceEndpointStallClear)(DRV_HANDLE handle, USB_ENDPOINT endpoint);	This is a pointer to the device endpoint stall clear function
bool (* deviceEndpointIsEnabled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);	This is pointer to the device endpoint enable status query function
bool (* deviceEndpointIsStalled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);	This is pointer to the device endpoint stall status query function
USB_ERROR (* deviceIRPSubmit)(DRV_HANDLE handle, USB_ENDPOINT endpoint, USB_DEVICE_IRP * irp);	This is a pointer to the device IRP submit function
USB_ERROR (* deviceIRPCancelAll)(DRV_HANDLE handle, USB_ENDPOINT endpoint);	This is a pointer to the device IRP Cancel all function
void (* deviceRemoteWakeupStart)(DRV_HANDLE handle);	This is a pointer to the device remote wakeup start function
void (* deviceRemoteWakeupStop)(DRV_HANDLE handle);	This is a pointer to the device remote wakeup stop function
USB_ERROR (* deviceTestModeEnter)(DRV_HANDLE handle, USB_TEST_MODE_SELECTORS testMode);	This is a pointer to the device Test mode enter function

Description

USB Driver Client Functions Interface (For Device Mode)

This structure is a group of function pointers pointing to the USB Driver Device Mode Client routines. The USB driver should export this group of functions so that the Device Layer can access the driver functionality.

Remarks

None.

DRV_USB_HOST_INTERFACE Structure

Group of function pointers to the USB Driver Host Mode Client Functions.

File

drv_usb.h

C

```
typedef struct {
  DRV_HANDLE (* open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
  void (* close)(DRV_HANDLE handle);
  void (* eventHandlerSet)(DRV_HANDLE handle, uintptr_t hReferenceData, DRV_USB_EVENT_CALLBACK
eventHandler);
  USB_ERROR (* hostIRPSubmit)(DRV_USB_HOST_PIPE_HANDLE pipeHandle, USB_HOST_IRP * irp);
  void (* hostIRPCancel)(USB_HOST_IRP * irp);
  bool (* hostEventsDisable)(DRV_HANDLE handle);
  void (* hostEventsEnable)(DRV_HANDLE handle, bool eventContext);
  DRV_USB_HOST_PIPE_HANDLE (* hostPipeSetup)(DRV_HANDLE client, uint8_t deviceAddress,
USB_ENDPOINT endpointAndDirection, uint8_t hubAddress, uint8_t hubPort, USB_TRANSFER_TYPE
pipeType, uint8_t bInterval, uint16_t wMaxPacketSize, USB_SPEED speed);
  void (* hostPipeClose)(DRV_USB_HOST_PIPE_HANDLE pipeHandle);
  DRV_USB_ROOT_HUB_INTERFACE rootHubInterface;
} DRV_USB_HOST_INTERFACE;
```

Members

Members	Description
DRV_HANDLE (* open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);	This is a pointer to the driver open function
void (* close)(DRV_HANDLE handle);	This is pointer to the driver close function
void (* eventHandlerSet)(DRV_HANDLE handle, uintptr_t hReferenceData, DRV_USB_EVENT_CALLBACK eventHandler);	This is a pointer to the event call back set function
USB_ERROR (* hostIRPSubmit)(DRV_USB_HOST_PIPE_HANDLE pipeHandle, USB_HOST_IRP * irp);	This is a pointer to the host IRP submit function
void (* hostIRPCancel)(USB_HOST_IRP * irp);	This is a pointer to the host IRP Cancel all function
bool (* hostEventsDisable)(DRV_HANDLE handle);	This is pointer to the host event disable function
void (* hostEventsEnable)(DRV_HANDLE handle, bool eventContext);	This is a pointer to the host event enable function
DRV_USB_HOST_PIPE_HANDLE (* hostPipeSetup)(DRV_HANDLE client, uint8_t deviceAddress, USB_ENDPOINT endpointAndDirection, uint8_t hubAddress, uint8_t hubPort, USB_TRANSFER_TYPE pipeType, uint8_t bInterval, uint16_t wMaxPacketSize, USB_SPEED speed);	This is a pointer to the host pipe setup function
void (* hostPipeClose)(DRV_USB_HOST_PIPE_HANDLE pipeHandle);	This is a pointer to the host pipe close function
DRV_USB_ROOT_HUB_INTERFACE rootHubInterface;	This is a pointer to the host Root Hub functions

Description

USB Driver Client Functions Interface (For Host Mode)

This structure is a group of function pointers pointing to the USB Driver Host Mode Client routines. The USB driver should export this group of functions so that the Host layer can access the driver functionality.

Remarks

None.

DRV_USB_OPMODE Enumeration

Possible operation modes of the USB Driver.

File

drv_usb.h

C

```
typedef enum {
  DRV_USB_OPMODE_DUAL_ROLE,
  DRV_USB_OPMODE_DEVICE,
  DRV_USB_OPMODE_HOST,
  DRV_USB_OPMODE_OTG
} DRV_USB_OPMODE;
```

Members

Members	Description
DRV_USB_OPMODE_DUAL_ROLE	The driver should be able to switch between Host and Device operation
DRV_USB_OPMODE_DEVICE	The driver should support device mode operation only
DRV_USB_OPMODE_HOST	The driver should support host mode operation only
DRV_USB_OPMODE_OTG	The driver should support the USB OTG protocol

Description

USB Driver Operation Mode enumeration

This enumeration lists the possible USB Driver Operation modes.

Remarks

None.

DRV_USB_ROOT_HUB_INTERFACE Structure

Group of function pointers to the USB Root Hub Functions.

File

drv_usb.h

C

```
typedef struct {
  USB_SPEED (* rootHubSpeedGet)(DRV_HANDLE handle);
  uint8_t (* rootHubPortNumbersGet)(DRV_HANDLE handle);
  uint32_t (* rootHubMaxCurrentGet)(DRV_HANDLE handle);
  void (* rootHubOperationEnable)(DRV_HANDLE handle, bool enable);
  bool (* rootHubOperationIsEnabled)(DRV_HANDLE handle);
  void (* rootHubInitialize)(DRV_HANDLE handle, USB_HOST_DEVICE_OBJ_HANDLE usbHostDeviceInfo);
  USB_HUB_INTERFACE rootHubPortInterface;
} DRV_USB_ROOT_HUB_INTERFACE;
```

Members

Members	Description
USB_SPEED (* rootHubSpeedGet)(DRV_HANDLE handle);	This function returns the bus speed of the root hub
uint8_t (* rootHubPortNumbersGet)(DRV_HANDLE handle);	Returns the number of ports that the root hub contains
uint32_t (* rootHubMaxCurrentGet)(DRV_HANDLE handle);	Returns the total current (in mA) that the root hub can supply
void (* rootHubOperationEnable)(DRV_HANDLE handle, bool enable);	Enables operation of the root hub
bool (* rootHubOperationIsEnabled)(DRV_HANDLE handle);	Return the status of the operation enable function
void (* rootHubInitialize)(DRV_HANDLE handle, USB_HOST_DEVICE_OBJ_HANDLE usbHostDeviceInfo);	This is the root hub initialize function
USB_HUB_INTERFACE rootHubPortInterface;	An interface to the Root Hub Port Control functions

Description

USB Root Hub API Interface

This structure is a group of function pointers pointing to the USB Driver Root Hub API routines. The USB Driver Root Hub should export this group of functions so that the Host layer can access the port functionality. The interface to the Root Hub APIs is offered through the USB Driver Host API.

Remarks

None.

DRV_USB_DEVICE_ENDPOINT_ALL Macro

File

drv_usb.h

C

```
#define DRV_USB_DEVICE_ENDPOINT_ALL 16
```

Description

This is macro DRV_USB_DEVICE_ENDPOINT_ALL.

Files

Files

Name	Description
drv_usb.h	USB Module Driver Interface File
drv_usb_config_template.h	USB driver configuration definitions template

Description

This section lists the source and header files used by the USB Driver Library.

drv_usb.h

USB Module Driver Interface File

Enumerations

	Name	Description
	DRV_USB_EVENT	Identifies the different events that the USB Driver provides.
	DRV_USB_OPMODE	Possible operation modes of the USB Driver.

Macros

	Name	Description
	DRV_USB_DEVICE_ENDPOINT_ALL	This is macro DRV_USB_DEVICE_ENDPOINT_ALL.
	DRV_USB_HOST_PIPE_HANDLE_INVALID	Defines the USB Driver Host Pipe Invalid handle

Structures

	Name	Description
	DRV_USB_DEVICE_INTERFACE	Group of function pointers to the USB Driver Device Mode Client Functions.
	DRV_USB_HOST_INTERFACE	Group of function pointers to the USB Driver Host Mode Client Functions.
	DRV_USB_ROOT_HUB_INTERFACE	Group of function pointers to the USB Root Hub Functions.

Types

	Name	Description
	DRV_USB_EVENT_CALLBACK	Type of the USB event callback function
	DRV_USB_HOST_PIPE_HANDLE	Defines the USB Driver Host Pipe handle type

Description

PIC32 USB Module Driver Interface Header File

This file describes the interface that any USB module driver must implement in order for it to work with MPLAB Harmony USB Host and Device Stack.

File Name

drv_usb.h

Company

Microchip Technology Inc.

drv_usb_config_template.h

USB driver configuration definitions template

Macros

	Name	Description
	DRV_USB_INDEX	This is macro DRV_USB_INDEX.
	DRV_USB_INTERRUPT_SOURCE	This is macro DRV_USB_INTERRUPT_SOURCE.
	DRV_USB_PERIPHERAL_ID	This is macro DRV_USB_PERIPHERAL_ID.

Description

USB Driver Configuration Definitions for the template version These definitions statically define the driver's mode of operation.

File Name

drv_usb_config_template.h

Company

Microchip Technology Inc.

MRF24W Wi-Fi Driver Library

This topic describes the MRF24W Wi-Fi Driver Library.

Introduction

This library provides a low-level abstraction of the MRF24W Wi-Fi Driver Library that is available on the Microchip family of microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, there by hiding differences from one microcontroller variant to another.

Description

The Wi-Fi software library, in conjunction with the MRF24WG0MA module, allows an application to:

- Join an existing 802.11 Wi-Fi network
- Create a 802.11 Wi-Fi network

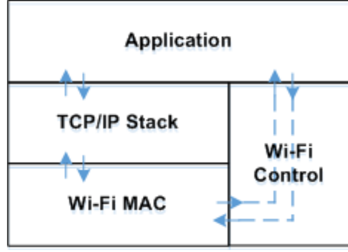
Note: The MRF24WG0MA/B module has an Over-the-Air (OTA) mechanism for updating the firmware utilizing code in the pic32_eth_wifi_web_server demonstration. Refer to Updating MRF24WG0MA/B Firmware for details. **Only the even numbered firmware versions work with MPLAB Harmony.**

The following application services are provided by the Wi-Fi library:

- Configuring Wi-Fi connection (SSID, security mode, channel list, etc.)
- Join an existing network or create an "AdHoc" Wi-Fi network
- Scan for other Wi-Fi devices in the area
- Getting Wi-Fi network status
- Wi-Fi power control

The MAC layer services are not directly accessible to the application; this portion of the code resides under the TCP/IP Stack MAC module software layers and is used by stack services to transmit and receive data over a Wi-Fi network. The following diagram shows the interaction of the primary software blocks in a Wi-Fi application.

Wi-Fi Software Block Diagram



Using the Library

This topic describes the basic architecture of the MRF24W Wi-Fi Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_wifi.h`

The interface to the MRF24W Wi-Fi Driver Library is defined in the `drv_wifi.h` header file.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the Wi-Fi module on Microchip's microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The Wi-Fi Library provides the following functionality:

- Wi-Fi library initialization
- Wi-Fi network configuration
- Wi-Fi network connection
- Scanning for existing Wi-Fi networks
- Wi-Fi event processing
- Wi-Fi status

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The Library Interface functions are divided into various sub-sections, which address one of the blocks or the overall operation of the Wi-Fi module.

Library Interface Section	Description
Wi-Fi Initialization Functions	This section provides functions that initialize the Wi-Fi library and allow its API to be used.
Wi-Fi Network General Configuration Functions	This section provides functions that configure the Wi-Fi interface for use.
Wi-Fi Network Security Configuration Functions	This section provides functions that configure the security mode of the Wi-Fi connection.
Wi-Fi Network Power Configuration Functions	This section provides functions that configure max TX power, as well as power-saving modes during a connection.
Wi-Fi Network Multicast Filter Configuration Functions	This section provides functions that configure the MRF24WG module to receive multicast address packets.

Wi-Fi Network Connection Functions	This section provides functions that begin the connection and disconnection processes.
Wi-Fi Network Gratuitous ARP Configuration Functions	This section provides functions that start a gratuitous ARP to maintain a connection with an Access Point when no other data traffic is occurring.
Wi-Fi Network Configuration - Other Functions	This section provides additional miscellaneous functions for configuring the Wi-Fi connection.
Wi-Fi Scanning Functions	This section provides functions that initiate a Wi-Fi scan and allow retrieval of the scan results.
Wi-Fi Event Processing Functions	This section provides callback functions that inform the application of Wi-Fi related events.
Wi-Fi Data Configuration Functions	This section provides functions that save and restore Wi-Fi configuration data.
Wi-Fi Status Functions	This section provides functions that retrieve the Wi-Fi connection status.

How the Library Works

This section describes how the Wi-Fi Driver Library operates.

Description

Before the driver is ready for use, its should be configured (compile time configuration). Refer to the Configuring the Library section for more details on how to configure the driver.

There are few run-time configuration items that are done during initialization of the driver instance, and a few that are client-specific and are done using dedicated functions.

To use the Wi-Fi Driver, initialization and client functions should be invoked in a specific sequence to ensure correct operation.

System Initialization

This section describes initialization and reinitialization features.

Description

Wi-Fi initialization configures the MRF24WG module and then directs it to join (or create) a Wi-Fi network. The MRF24WG module defaults to open security and scans all channels in the domain. Therefore, to initialize and connect with the minimum function call overhead in an open security network, the following functions can be used:

```
DRV_WIFI_SsidSet("MySsidName");
DRV_WIFI_Connect();           // start the connection process
```

Alternatively, the following functions could be used to achieve the same effect:

```
DRV_WIFI_ChannelListSet(0);   // 0 = all channels in domain
DRV_WIFI_NetworkTypeSet(DRV_WIFI_NETWORK_TYPE_INFRASTRUCTURE);
DRV_WIFI_SecurityOpenSet(DRV_WIFI_SECURITY_OPEN);
DRV_WIFI_SsidSet("MySsidName");
DRV_WIFI_Connect();
```

Client Functionality

This section describes core operation.

Description

From the client perspective, once Wi-Fi initialization is complete and the connection process has started, the client responds to Wi-Fi events. The client is notified of events by the callback function DRV_WIFI_ProcessEvent. The parameters into that function are `event` and `eventInfo`, where `event` is the event code and `eventInfo` is additional information about the event.

Wi-Fi Connection Events

Event Type	Description
DRV_WIFI_EVENT_DISCONNECT_DONE	A Wi-Fi connection attempt has completed successfully
DRV_WIFI_EVENT_CONNECTION_FAILED	A Wi-Fi connection attempt has failed
DRV_WIFI_EVENT_CONNECTION_TEMPORARILY_LOST	An existing Wi-Fi connection has been temporarily lost (and MRF24WG module is attempting to reconnect)
DRV_WIFI_EVENT_CONNECTION_PERMANENTLY_LOST	An existing Wi-Fi connection has been permanently lost
DRV_WIFI_EVENT_CONNECTION_REESTABLISHED	A connection that was temporarily lost has been regained
DRV_WIFI_EVENT_DISCONNECT_DONE	An application disconnect request has completed

Scan Events

Event Type	Description
DRV_WIFI_EVENT_SCAN_RESULTS_READY	The requested scan is complete and scan results are ready to be read

Key Events

Event Type	Description
DRV_WIFI_EVENT_KEY_CALCULATION_REQUEST	The requested key calculation is complete and the binary key is ready to send to the MRF24WG module

Error Events

Event Type	Description
DRV_WIFI_EVENT_ERROR	A Wi-Fi driver error has occurred

For some of the event types previously listed, there is an `eventInfo` parameter that provides additional information about the event. Not all event types require the `eventInfo` parameter. The tables in DRV_WIFI_CONNECTION_TEMPORARILY_LOST provide additional detail about the `eventInfo` parameter for those events that use it.

DRV_WIFI_CONNECTION_TEMPORARILY_LOST

`eventInfo` is defined as:

15:8	7	6	5:0
Not used	Deauth	Disassoc	Event Data

Bits 15:8 are not used.

If Deauth = 1 or Disassoc = 1, Event Data is:

Table 1: Deauth or Disassoc Event Data

DRV_WIFI_UNSPECIFIED	1
DRV_WIFI_PREV_AUTH_NOT_VALID	2
DRV_WIFI_DISASSOC_DUE_TO_INACTIVITY	3
DRV_WIFI_DISASSOC_AP_BUSY	4
DRV_WIFI_CLASS2_FRAME_FROM_NONAUTH_STA	6
DRV_WIFI_CLASS3_FRAME_FROM_NONASSOC_STA	7
DRV_WIFI_DISASSOC_STA_HAS_LEFT	8
DRV_WIFI_STA_REQ_ASSOC_WITHOUT_AUTH	9
DRV_WIFI_INVALID_IE	13
DRV_WIFI_MIC_FAILURE	14
DRV_WIFI_4WAY_HANDSHAKE_TIMEOUT	15
DRV_WIFI_GROUP_KEY_HANDSHAKE_TIMEOUT	16
DRV_WIFI_IE_DIFFERENT	17
DRV_WIFI_INVALID_GROUP_CIPHER	18
DRV_WIFI_INVALID_PAIRWISE_CIPHER	19
DRV_WIFI_INVALID_AKMP	20
DRV_WIFI_UNSUPP_RSN_VERSION	21
DRV_WIFI_INVALID_RSN_IE_CAP	22
DRV_WIFI_IEEE8021X_FAILED	23
DRV_WIFI_CIPHER_SUITE_REJECTED	24

If both `Deauth` and `Disassoc` equal '0', Event Data is:

Table 2: Connection Lost Event Data

DRV_WIFI_CIPHER_SUITE_REJECTED	0
DRV_WIFI_BEACON_TIMEOUT	1
DRV_WIFI_DEAUTH_RECEIVED	2
DRV_WIFI_DISASSOCIATE_RECEIVED	3
DRV_WIFI_TKIP_MIC_FAILURE	4
DRV_WIFI_LINK_DOWN	5

DRV_WIFI_EVENT_CONNECTION_FAILED

`eventInfo` is defined as:

15:8	7:0
Status	Reason

Status is:

Table 1: Connection Failures

DRV_WIFI_JOIN_FAILURE	2
DRV_WIFI_AUTHENTICATION_FAILURE	3
DRV_WIFI_ASSOCIATION_FAILURE	4
DRV_WIFI_WEP_HANDSHAKE_FAILURE	5
DRV_WIFI_PSK_CALCULATION_FAILURE	6
DRV_WIFI_PSK_HANDSHAKE_FAILURE	7
DRV_WIFI_ADHOC_JOIN_FAILURE	8
DRV_WIFI_SECURITY_MISMATCH_FAILURE	9
DRV_WIFI_NO_SUITABLE_AP_FOUND_FAILURE	10
DRV_WIFI_RETRY_FOREVER_NOT_SUPPORTED_FAILURE	11
DRV_WIFI_LINK_LOST	12
DRV_WIFI_TKIP_MIC_FAILURE	13
DRV_WIFI_RSN_MIXED_MODE_NOT_SUPPORTED	14
DRV_WIFI_RECV_DEAUTH	15
DRV_WIFI_RECV_DISASSOC	16
DRV_WIFI_WPS_FAILURE	17
DRV_WIFI_P2P_FAILURE	18
DRV_WIFI_LINK_DOWN	19

If Status = DRV_WIFI_RECV_DEAUTH or Status = DRV_WIFI_RECV_DISASSOC, Reason is one of the values from **Table 2: Connection Lost Event Data** in DRV_WIFI_CONNECTION_TEMPORARILY_LOST.

If Status = DRV_WIFI_AUTHENTICATION_FAILURE or Status = DRV_WIFI_ASSOCIATION_FAILURE, Reason is:

Table 2: Authentication and Association Failures

DRV_WIFI_UNSPECIFIED_FAILURE	1
DRV_WIFI_CAPS_UNSUPPORTED	10
DRV_WIFI_REASSOC_NO_ASSOC	11
DRV_WIFI_ASSOC_DENIED_UNSPEC	12
DRV_WIFI_NOT_SUPPORTED_AUTH_ALG	13
DRV_WIFI_UNKNOWN_AUTH_TRANSACTION	14
DRV_WIFI_CHALLENGE_FAIL	15
DRV_WIFI_AUTH_TIMEOUT	16
DRV_WIFI_AP_UNABLE_TO_HANDLE_NEW_STA	17
DRV_WIFI_ASSOC_DENIED_RATES	18
DRV_WIFI_ASSOC_DENIED_NOSHORTPREAMBLE	19
DRV_WIFI_ASSOC_DENIED_NOPBCC	20
DRV_WIFI_ASSOC_DENIED_NOAGILITY	21

DRV_WIFI_ASSOC_DENIED_NOSHORTTIME	25
DRV_WIFI_ASSOC_DENIED_NODSSSOFDM	26
DRV_WIFI_NOT_VALID_IE	40
DRV_WIFI_NOT_VALID_GROUPCIPHER	41
DRV_WIFI_NOT_VALID_PAIRWISE_CIPHER	42
DRV_WIFI_NOT_VALID_AKMP	43
DRV_WIFI_UNSUPPORTED_RSN_VERSION	44
DRV_WIFI_INVALID_RSN_IE_CAP	45
DRV_WIFI_CIPHER_SUITE_REJECTED	46
DRV_WIFI_TIMEOUT	47

If Status = DRV_WIFI_WPS_FAILURE, Reason is defined as:

7:4	3:0
wpsState	wpsConfigErr

Table 3: wpsState

DRV_WIFI_EAPOL_START	1
DRV_WIFI_EAP_REQ_IDENTITY	2
DRV_WIFI_EAP_RSP_IDENTITY	3
DRV_WIFI_EAP_WPS_START	4
DRV_WIFI_EAP_RSP_M1	5
DRV_WIFI_EAP_REQ_M2	6
DRV_WIFI_EAP_RSP_M3	7
DRV_WIFI_EAP_REQ_M4	8
DRV_WIFI_EAP_RSP_M5	9
DRV_WIFI_EAP_REQ_M6	10
DRV_WIFI_EAP_RSP_M7	11
DRV_WIFI_EAP_REQ_M8	12
DRV_WIFI_EAP_RSP_DONE	13
DRV_WIFI_EAP_RSP_DONE	14

Table 4: wpsConfigErr

DRV_WIFI_WPS_NOERR	0
DRV_WIFI_WPS_SESSION_OVERLAPPED	1
DRV_WIFI_WPS_SESSION_OVERLAPPED	2
DRV_WIFI_WPS_24G_NOT_SUPPORTED	3
DRV_WIFI_WPS_24G_NOT_SUPPORTED	4
DRV_WIFI_WPS_INVALID_MSG	5
DRV_WIFI_WPS_AUTH_FAILURE	6
DRV_WIFI_WPS_ASSOC_FAILURE	7
DRV_WIFI_WPS_MSG_TIMEOUT	8

DRV_WIFI_WPS_SESSION_TIMEOUT	9
DRV_WIFI_WPS_DEVPASSWD_AUTH_FAILURE	10
DRV_WIFI_WPS_NO_CONN_TOREG	11
DRV_WIFI_WPS_MULTI_PBC_DETECTED	12
DRV_WIFI_WPS_EAP_FAILURE	13
DRV_WIFI_WPS_DEV_BUSY	14
DRV_WIFI_WPS_SETUP_LOCKED	15

If Status = DRV_WIFI_P2P_FAILURE, Reason is defined as:

7:4	3:0
p2pState	p2pError

Table 5: p2pState

DRV_WIFI_P2P_IDLE	0
DRV_WIFI_P2P_SCAN	1
DRV_WIFI_P2P_LISTEN	2
DRV_WIFI_P2P_FIND	3
DRV_WIFI_P2P_START_FORMATION	4
DRV_WIFI_P2P_NEG_REQ_DONE	5
DRV_WIFI_P2P_WAIT_NEG_REQ_DONE	6
DRV_WIFI_P2P_WAIT_FORMATION_DONE	7
DRV_WIFI_P2P_INVITE	8
DRV_WIFI_P2P_PROVISION	9
DRV_WIFI_P2P_CLIENT	10

Table 6: p2pError

DRV_WIFI_WFD_SUCCESS	0
DRV_WIFI_WFD_INFO_CURRENTLY_UNAVAILABLE	1
DRV_WIFI_WFD_INCOMPATIBLE_PARAMS	2
DRV_WIFI_WFD_LIMIT_REACHED	3
DRV_WIFI_WFD_INVALID_PARAMS	4
DRV_WIFI_WFD_UNABLE_TO_ACCOMMODATE	5
DRV_WIFI_WFD_PREV_PROTOCOL_ERROR	6
DRV_WIFI_WFD_NO_COMMON_CHANNELS	7
DRV_WIFI_WFD_UNKNOWN_GROUP	8
DRV_WIFI_WFD_INCOMPATIBLE_PROV_METHOD	10
DRV_WIFI_WFD_REJECTED_BY_USER	11
DRV_WIFI_WFD_NO_MEM	12
DRV_WIFI_WFD_INVALID_ACTION	13
DRV_WIFI_WFD_TX_FAILURE	14
DRV_WIFI_WFD_TIME_OUT	15

Configuring the Library

The configuration of the MRF24W Wi-Fi driver is based on the file `drv_wifi_config.h`.

This header file contains the configuration selection for the Wi-Fi Driver. Based on the selections made, the Wi-Fi Driver may support the selected features. These configuration settings will apply to all instances of the Wi-Fi Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Overview section for more details.

Sample Functionality

Please refer to the Wi-Fi demos for examples of how to use the various features in the MRF24W Wi-Fi Driver Library.

Building the Library

This section lists the files that are available in the MRF24W Wi-Fi Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/wifi/mrf24w`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
/drv_wifi.h	Contains all data types, define constants, and function prototypes for interfacing to the Wi-Fi driver

Required File(s)

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
drv_wifi_com.c	MRF24W Driver com layer (specific to the MRF24WG). Provides access to the MRF24W Wi-Fi controller.
drv_wifi_commands.c	MRF24W commands (based on system commander) implementation. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_commands.h	Wi-Fi MAC interface functions. Wi-Fi-specific MAC function prototypes called by the TCP/IP stack.
drv_wifi_config_data.c	MRF24W configuration data. Stores and retrieves Wi-Fi configuration information to Non-volatile Memory (NVM).
drv_wifi_config_data.h	Wi-Fi MAC interface functions. Wi-Fi-specific MAC function prototypes called by the TCP/IP stack.
drv_wifi_connect.c	MRF24W connection support. Functions in this module support the connection process for the MRF24W.
drv_wifi_connection_algorithm.c	MRF24W Driver connection algorithm. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_connection_manager.c	MRF24W Driver connection manager. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_connection_profile.c	MRF24W Driver connection profile. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_debug_output.c	MRF24W Driver Wi-Fi console. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_debug_output.h	Wi-Fi MAC interface functions. Wi-Fi-specific MAC function prototypes called by the TCP/IP stack.
drv_wifi_easy_config.c	MRF24W Driver. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_easy_config.h	Wi-Fi MAC interface functions. Wi-Fi-specific MAC function prototypes called by the TCP/IP stack.

drv_wifi_eint.c	MRF24W Driver feature to support external interrupts.
drv_wifi_eint.h	MRF24W external interrupt header file. MRF24W specific interrupt functions.
drv_wifi_event_handler.c	MRF24W Driver event handler. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_events.c	TCP/IP MRF24W MAC events implementation. Processes Wi-Fi events via callback functions.
drv_wifi_init.c	MRF24W Driver initialization. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_mac.c	MRF24W Driver Medium Access Control (MAC) layer. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_mac.h	Wi-Fi MAC interface functions. Module for Microchip TCP/IP Stack PIC32 implementation for multiple MAC support.
drv_wifi_mac_pic32.c	MRF24W Driver Medium Access Control (MAC) layer. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_mgmt_msg.c	MRF24W Driver management messages (specific to the MRF24WG). Provides access to the MRF24W Wi-Fi controller.
drv_wifi_mgmt_msg.h	Wi-Fi MAC interface functions. Wi-Fi-specific MAC function prototypes called by the TCP/IP stack.
drv_wifi_param_msg.c	MRF24W Driver management Set/Get parameter messages (specific to the MRF24WG). Provides access to the MRF24W Wi-Fi controller.
drv_wifi_pbkdf2.c	MRF24W WPA supplicant. SHA1 Hash implementation and interface functions.
drv_wifi_power_save.c	MRF24W Driver power-saving functions. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_priv.h	Wi-Fi MAC interface functions. Wi-Fi-specific MAC function prototypes called by the TCP/IP stack.
drv_wifi_raw.c	MRF24W RAW Driver. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_raw.h	Wi-Fi MAC interface functions. Wi-Fi-specific MAC function prototypes called by the TCP/IP stack.
drv_wifi_scan.c	MRF24W Driver Scan functions. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_spi.c	MRF24W SPI Driver. Supports SPI communications to the MRF24W module.
drv_wifi_spi.h	Wi-Fi MAC interface functions. Wi-Fi-specific MAC function prototypes called by the TCP/IP stack.
drv_wifi_spi_init.c	MRF24W Driver SPI interface routines. Initializes the SPI hardware used to communicate with the MRF24W.
drv_wifi_tx_power.c	MRF24W Driver Transmit (TX) Power functions. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_update_firmware_common.c	MRF24W Driver Medium Access Control (MAC) layer. Provides access to the MRF24W Wi-Fi controller.
drv_wifi_update_firmware_tcpclient_txt.c	MRF24W Driver Medium Access Control (MAC) layer. Provides access to the MRF24W Wi-Fi controller.

drv_wifi_update_firmware_uart.c	MRF24W Driver Medium Access Control (MAC) layer. Provides access to the MRF24W Wi-Fi controller.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files.

Module Dependencies

The MRF24W Wi-Fi Driver Library depends on the following modules:

- SPI Driver Library
- NVM Driver Library
- Operating System Abstraction Layer (OSAL) Library Help
- Clock System Service Library
- System Service Library Introduction
- Console System Service Library
- File System Service Library
- Interrupt System Service Library
- Timer System Service Library
- Debug System Service Library
- Ports System Service Library

Library Interface

a) Wi-Fi Initialization Functions

	Name	Description
	DRV_WIFI_Deinitialize	Initializes the MRF24WG Wi-Fi driver. **Implementation:** Dynamic
	DRV_WIFI_Initialize	Initializes the MRF24WG Wi-Fi driver. **Implementation:** Dynamic
	DRV_WIFI_MRF24W_ISR	Wi-Fi driver (MRF24WG specific) interrupt service routine. **Implementation:** Dynamic
	DRV_WIFI_ConnectStateMachine	Starts the state machine to connect to an 802.11 network. **Implementation:** Dynamic

b) Wi-Fi Network General Configuration Functions

	Name	Description
	DRV_WIFI_ChannelListSet	Sets the channel list. **Implementation:** Dynamic
	DRV_WIFI_NetworkTypeSet	Sets the Wi-Fi network type. **Implementation:** Dynamic
	DRV_WIFI_ReconnectModeSet	Sets the Wi-Fi reconnection mode. **Implementation:** Dynamic
	DRV_WIFI_SsidSet	Sets the SSID. **Implementation:** Dynamic
	DRV_WIFI_AdhocContextSet	Sets the AdHoc context. **Implementation:** Dynamic

c) Wi-Fi Network Security Configuration Functions

	Name	Description
	DRV_WIFI_SetPSK	Sets the binary WPA PSK code in WPS. **Implementation:** Dynamic
	DRV_WIFI_SecurityOpenSet	Sets Wi-Fi security to open (no security). **Implementation:** Dynamic
	DRV_WIFI_SecurityWepSet	Sets Wi-Fi security to use WEP. **Implementation:** Dynamic
	DRV_WIFI_SecurityWpaSet	Sets Wi-Fi security to use WPA or WPA2. **Implementation:** Dynamic
	DRV_WIFI_SecurityWpsSet	Sets Wi-Fi security to use WPS. **Implementation:** Dynamic

d) Wi-Fi Network Power Configuration Functions

	Name	Description
	DRV_WIFI_HibernateEnable	Puts the MRF24WG into hibernate mode. **Implementation:** Dynamic
	DRV_WIFI_PsPollDisable	Disables PS-Poll mode. **Implementation:** Dynamic
	DRV_WIFI_PsPollEnable	Enables PS Poll mode. **Implementation:** Dynamic

e) Wi-Fi Network Multicast Filter Configuration Functions

	Name	Description
	DRV_WIFI_HWMulticastFilterSet	Sets a multicast address filter using one of the two hardware multicast filters. **Implementation:** Dynamic
	DRV_WIFI_SWMulticastFilterSet	Sets a multicast address filter using one of the software multicast filters. **Implementation:** Dynamic

f) Wi-Fi Network Connection Functions

	Name	Description
	DRV_WIFI_Connect	Directs the MRF24WG to connect to a Wi-Fi network. **Implementation:** Dynamic
	DRV_WIFI_Disconnect	Directs the MRF24WG to disconnect from a Wi-Fi network. **Implementation:** Dynamic

g) Wi-Fi Network Gratuitous ARP Configuration Functions

	Name	Description
	DRV_WIFI_GratuitousArpStart	Starts a periodic gratuitous ARP response. **Implementation:** Dynamic
	DRV_WIFI_GratuitousArpStop	Stops a periodic gratuitous ARP. **Implementation:** Dynamic

h) Wi-Fi Network Configuration - Other Functions

	Name	Description
	DRV_WIFI_BssidSet	Sets the Basic Service Set Identifier (BSSID). **Implementation:** Dynamic
	DRV_WIFI_SetLinkDownThreshold	Sets number of consecutive Wi-Fi TX failures before link is considered down. **Implementation:** Dynamic
	DRV_WIFI_SWMultiCastFilterEnable	Forces the MRF24WG to use software multicast filters instead of hardware multicast filters. **Implementation:** Dynamic
	DRV_WIFI_MacAddressSet	Uses a different MAC address for the MRF24W. **Implementation:** Dynamic
	DRV_WIFI_RtsThresholdSet	Sets the RTS Threshold. **Implementation:** Dynamic
	DRV_WIFI_ScanContextSet	Sets the Wi-Fi scan context. **Implementation:** Dynamic
	DRV_WIFI_TxModeSet	Configures 802.11 TX mode. **Implementation:** Dynamic
	DRV_WIFI_RssiSet	Sets RSSI restrictions when connecting. **Implementation:** Dynamic
	DRV_WIFI_EasyConfigTask_RtosTask	Implements Wi-Fi driver easy configuration RTOS task. **Implementation:** Dynamic
	DRV_WIFI_InitStateMachine_RtosTask	Implements Wi-Fi driver initialization RTOS task. **Implementation:** Dynamic
	DRV_WIFI_ISR_RtosTask	Implements Wi-Fi driver ISR RTOS task. **Implementation:** Dynamic

© 2013-2015 Microchip Technology Inc.

	DRV_WIFI_ISR_SemUnlock	Unlocks semaphore in Wi-Fi RTOS ISR. **Implementation:** Dynamic
	DRV_WIFI_MACProcess_RtosTask	Implements Wi-Fi driver MAC process RTOS task. **Implementation:** Dynamic
	DRV_WIFI_TASK_MUTEX_Lock	Locks MUTEX in Wi-Fi RTOS task when necessary. **Implementation:** Dynamic
	DRV_WIFI_TASK_MUTEX_Unlock	Unlocks MUTEX in Wi-Fi RTOS task. **Implementation:** Dynamic
	DRV_WIFI_RSSI_Cache_FromRxDataRead	Caches RSSI value from Rx data packet. **Implementation:** Dynamic
	DRV_WIFI_RSSI_Get_FromRxDataRead	Reads RSSI value from Rx data packet. **Implementation:** Dynamic
	DRV_WIFI_INT_Handle	Wi-Fi driver interrupt handle. **Implementation:** Dynamic
	DRV_WIFI_ISR_SemLock	Locks semaphore in Wi-Fi RTOS ISR. **Implementation:** Dynamic
	DRV_WIFI_RTOS_TaskInit	Initializes RTOS tasks for Wi-Fi driver. **Implementation:** Dynamic
	DRV_WIFI_SpiClose	Closes SPI object for Wi-Fi driver. **Implementation:** Dynamic
	DRV_WIFI_SpiInit	Initializes SPI object for Wi-Fi driver. **Implementation:** Dynamic

i) Wi-Fi Scanning Functions

	Name	Description
	DRV_WIFI_Scan	Commands the MRF24W to start a scan operation. This will generate the WF_EVENT_SCAN_RESULTS_READY event. **Implementation:** Dynamic
	DRV_WIFI_ScanGetResult	Read selected scan results back from MRF24W. **Implementation:** Dynamic
	DRV_WIFI_ScanContextGet	Gets the Wi-Fi scan context. **Implementation:** Dynamic

j) Wi-Fi Event Processing Functions

	Name	Description
	DRV_WIFI_ProcessEvent	Processes Wi-Fi event. **Implementation:** Dynamic
	DRV_WIFI_SoftApEventInfoGet	Gets the stored Soft AP event info. **Implementation:** Dynamic
	DRV_WIFI_SoftAPContextSet	Sets the SoftAP context. **Implementation:** Dynamic

k) Wi-Fi Data Configuration Functions

	Name	Description
	DRV_WIFI_ConfigDataErase	Erases configuration data from the board EEPROM. **Implementation:** Dynamic
	DRV_WIFI_ConfigDataLoad	Loads configuration data from the board EEPROM. **Implementation:** Dynamic

	DRV_WIFI_ConfigDataPrint	Outputs to console the configuration data from the board EEPROM. **Implementation:** Dynamic
	DRV_WIFI_ConfigDataSave	Save configuration data to the board EEPROM. **Implementation:** Dynamic

l) Wi-Fi Status Functions

	Name	Description
	DRV_WIFI_BssidGet	Gets the BSSID set in DRV_WIFI_BssidSet(). **Implementation:** Dynamic
	DRV_WIFI_ChannelListGet	Gets the channel list. **Implementation:** Dynamic
	DRV_WIFI_ConnectContextGet	Gets the current Wi-Fi connection context. **Implementation:** Dynamic
	DRV_WIFI_ConnectionStateGet	Gets the current Wi-Fi connection state. **Implementation:** Dynamic
	DRV_WIFI_DeviceInfoGet	Retrieves MRF24WG device information. **Implementation:** Dynamic
	DRV_WIFI_HWMulticastFilterGet	Gets a multicast address filter from one of the two multicast filters. **Implementation:** Dynamic
	DRV_WIFI_MacAddressGet	Retrieves the MRF24WG MAC address. **Implementation:** Dynamic
	DRV_WIFI_MacStatsGet	Gets MAC statistics. **Implementation:** Dynamic
	DRV_WIFI_NetworkTypeGet	Gets the Wi-Fi network type. **Implementation:** Dynamic
	DRV_WIFI_PowerSaveStateGet	Gets the current power-saving state.
	DRV_WIFI_ReconnectModeGet	Gets the Wi-Fi reconnection mode. **Implementation:** Dynamic
	DRV_WIFI_RegionalDomainGet	Retrieves the MRF24WG Regional domain. **Implementation:** Dynamic
	DRV_WIFI_RtsThresholdGet	Gets the RTS Threshold. **Implementation:** Dynamic
	DRV_WIFI_SecurityGet	Gets the current Wi-Fi security setting. **Implementation:** Dynamic
	DRV_WIFI_SsidGet	Gets the SSID. **Implementation:** Dynamic
	DRV_WIFI_TxModeGet	Gets 802.11 TX mode. **Implementation:** Dynamic
	DRV_WIFI_WepKeyTypeGet	Gets the WEP Key type. **Implementation:** Dynamic
	DRV_WIFI_WPSCredentialsGet	Gets the WPS credentials. **Implementation:** Dynamic
	DRV_WIFI_RssiGet	Gets RSSI value set in DRV_WIFI_RssiSet(). **Implementation:** Dynamic
	DRV_WIFI_isHibernateEnable	Checks if MRF24W is in hibernate mode. **Implementation:** Dynamic
	DRV_WIFI_SecurityTypeGet	This is function DRV_WIFI_SecurityTypeGet.
	DRV_WIFI_TxPowerFactoryMaxGet	Retrieves the factory-set max TX power from the MRF24W. **Implementation:** Dynamic

	DRV_WIFI_TxPowerMaxGet	Gets the TX max power on the MRF24WG0M. **Implementation:** Dynamic
	DRV_WIFI_TxPowerMaxSet	Sets the TX max power on the MRF24WG0M. **Implementation:** Dynamic

m) Data Types and Constants

	Name	Description
	DRV_WIFI_BSSID_LENGTH	This is macro DRV_WIFI_BSSID_LENGTH.
	DRV_WIFI_DEAUTH_REASONCODE_MASK	This is macro DRV_WIFI_DEAUTH_REASONCODE_MASK.
	DRV_WIFI_DEFAULT_ADHOC_BEACON_PERIOD	ms
	DRV_WIFI_DEFAULT_ADHOC_HIDDEN_SSID	Default values for Wi-Fi AdHoc settings
	DRV_WIFI_DEFAULT_ADHOC_MODE	This is macro DRV_WIFI_DEFAULT_ADHOC_MODE.
	DRV_WIFI_DEFAULT_PS_DTIM_ENABLED	DTIM wake-up enabled (normally the case)
	DRV_WIFI_DEFAULT_PS_DTIM_INTERVAL	number of beacon periods
	DRV_WIFI_DEFAULT_PS_LISTEN_INTERVAL	100ms multiplier, e.g. 1 * 100ms = 100ms
	DRV_WIFI_DEFAULT_SCAN_COUNT	Default values for Wi-Fi scan context
	DRV_WIFI_DEFAULT_SCAN_MAX_CHANNEL_TIME	ms
	DRV_WIFI_DEFAULT_SCAN_MIN_CHANNEL_TIME	ms
	DRV_WIFI_DEFAULT_SCAN_PROBE_DELAY	us
	DRV_WIFI_DEFAULT_WEP_KEY_TYPE	This is macro DRV_WIFI_DEFAULT_WEP_KEY_TYPE.
	DRV_WIFI_DISABLED	
	DRV_WIFI_DISASSOC_REASONCODE_MASK	This is macro DRV_WIFI_DISASSOC_REASONCODE_MASK.
	DRV_WIFI_ENABLED	This is macro DRV_WIFI_ENABLED.
	DRV_WIFI_MAX_CHANNEL_LIST_LENGTH	This is macro DRV_WIFI_MAX_CHANNEL_LIST_LENGTH.
	DRV_WIFI_MAX_NUM_RATES	This is macro DRV_WIFI_MAX_NUM_RATES.
	DRV_WIFI_MAX_SECURITY_KEY_LENGTH	This is macro DRV_WIFI_MAX_SECURITY_KEY_LENGTH.
	DRV_WIFI_MAX_SSID_LENGTH	This is macro DRV_WIFI_MAX_SSID_LENGTH.
	DRV_WIFI_MAX_WEP_KEY_LENGTH	This is macro DRV_WIFI_MAX_WEP_KEY_LENGTH.
	DRV_WIFI_MAX_WPA_PASS_PHRASE_LENGTH	must exclude string terminator
	DRV_WIFI_MIN_WPA_PASS_PHRASE_LENGTH	must exclude string terminator
	DRV_WIFI_NETWORK_TYPE_ADHOC	This is macro DRV_WIFI_NETWORK_TYPE_ADHOC.
	DRV_WIFI_NETWORK_TYPE_INFRASTRUCTURE	Selection of different Wi-Fi network types
	DRV_WIFI_NETWORK_TYPE_P2P	not supported

	DRV_WIFI_NETWORK_TYPE_SOFT_AP	This is macro DRV_WIFI_NETWORK_TYPE_SOFT_AP.
	DRV_WIFI_NO_ADDITIONAL_INFO	eventInfo define for DRV_WIFI_ProcessEvent() when no additional info is supplied
	DRV_WIFI_RETRY_ADHOC	This is macro DRV_WIFI_RETRY_ADHOC.
	DRV_WIFI_RETRY_FOREVER	This is macro DRV_WIFI_RETRY_FOREVER.
	DRV_WIFI_RTS_THRESHOLD_MAX	maximum RTS threshold size in bytes
	DRV_WIFI_SECURITY_EAP	not supported
	DRV_WIFI_SECURITY_OPEN	Selection of different Wi-Fi security types
	DRV_WIFI_SECURITY_WEP_104	This is macro DRV_WIFI_SECURITY_WEP_104.
	DRV_WIFI_SECURITY_WEP_40	This is macro DRV_WIFI_SECURITY_WEP_40.
	DRV_WIFI_SECURITY_WPA_AUTO_WITH_KEY	This is macro DRV_WIFI_SECURITY_WPA_AUTO_WITH_KEY.
	DRV_WIFI_SECURITY_WPA_AUTO_WITH_PASS_PHRASE	This is macro DRV_WIFI_SECURITY_WPA_AUTO_WITH_PASS_PHRASE.
	DRV_WIFI_SECURITY_WPS_PIN	This is macro DRV_WIFI_SECURITY_WPS_PIN.
	DRV_WIFI_SECURITY_WPS_PUSH_BUTTON	This is macro DRV_WIFI_SECURITY_WPS_PUSH_BUTTON.
	DRV_WIFI_WEP104_KEY_LENGTH	4 keys of 13 bytes each
	DRV_WIFI_WEP40_KEY_LENGTH	4 keys of 5 bytes each
	DRV_WIFI_WPA_KEY_LENGTH	This is macro DRV_WIFI_WPA_KEY_LENGTH.
	DRV_WIFI_WPS_PIN_LENGTH	7 digits + checksum byte
	DRV_WIFI_ADHOC_MODES	Selection of different AdHoc connection modes
	DRV_WIFI_ADHOC_NETWORK_CONTEXT	Contains data pertaining to Wi-Fi AdHoc context
	DRV_WIFI_CONNECTION_CONTEXT	Contains data pertaining to MRF24WG connection context
	DRV_WIFI_CONNECTION_STATES	Wi-Fi Connection states
	DRV_WIFI_DEVICE_INFO	Contains data pertaining to MRF24WG device type and version number
	DRV_WIFI_DEVICE_TYPES	Codes for Wi-Fi device types
	DRV_WIFI_DOMAIN_CODES	Regional domain codes.
	DRV_WIFI_EVENT_CONN_TEMP_LOST_CODES	Selection of different codes when Wi-Fi connection is temporarily lost.
	DRV_WIFI_EVENT_INFO	Selection of different EventInfo types
	DRV_WIFI_EVENTS	Selections for events that can occur.
	DRV_WIFI_GENERAL_ERRORS	This is type DRV_WIFI_GENERAL_ERRORS.
	DRV_WIFI_HIBERNATE_STATES	Wi-Fi Hibernate states

	DRV_WIFI_MAC_STATS	Wi-Fi MIB states
	DRV_WIFI_MGMT_ERRORS	Error codes returned when a management message is sent to the MRF24W
	DRV_WIFI_MGMT_INDICATE_SOFT_AP_EVENT	Contains data pertaining to Wi-Fi Soft AP event
	DRV_WIFI_MULTICAST_FILTER_IDS	Selections for software Multicast filter IDs
	DRV_WIFI_MULTICAST_FILTERS	Selections for Software Multicast Filters.
	DRV_WIFI_POWER_SAVE_STATES	Wi-Fi Power-Saving states
	DRV_WIFI_PS_POLL_CONTEXT	Contains data pertaining to Wi-Fi PS-Poll context
	DRV_WIFI_REASON_CODES	Selection of different codes when a deauthorization or disassociation event has occurred.
	DRV_WIFI_RECONNECT_MODES	Selection of different Reconnection modes
	DRV_WIFI_SCAN_CONTEXT	Contains data pertaining to Wi-Fi scan context
	DRV_WIFI_SCAN_RESULT	Contains data pertaining to Wi-Fi scan results
	DRV_WIFI_SCAN_TYPES	Selection of different Wi-Fi scan types
	DRV_WIFI_SECURITY_CONTEXT	Contains data pertaining to Wi-Fi security.
	DRV_WIFI_SOFT_AP_EVENT_REASON_CODES	Wi-Fi Soft AP event reason codes
	DRV_WIFI_SOFT_AP_STATES	Wi-Fi Soft AP events
	DRV_WIFI_STATUS_CODES	Selection of different codes when Wi-Fi connection fails due to association or authentication failure.
	DRV_WIFI_SWMULTICAST_CONFIG	Contains data pertaining to Wi-Fi software multicast filter configuration
	DRV_WIFI_TX_MODES	Selections for Wi-Fi TX mode
	DRV_WIFI_WEP_CONTEXT	Contains data pertaining to Wi-Fi WEP context
	DRV_WIFI_WEP_KEY_TYPE	Selections for WEP key type when using WEP security.
	DRV_WIFI_WPA_CONTEXT	Contains data pertaining to Wi-Fi WPA.
	DRV_WIFI_WPA_KEY_INFO	Contains data pertaining to Wi-Fi WPA Key
	DRV_WIFI_WPS_AUTH_TYPES	Selection of WPS Authorization types
	DRV_WIFI_WPS_CONTEXT	Contains data pertaining to Wi-Fi WPS security.
	DRV_WIFI_WPS_CREDENTIAL	Contains data pertaining to Wi-Fi WPS Credentials
	DRV_WIFI_WPS_ENCODE_TYPES	Selection of WPS Encoding types
	adhocMode	Selection of different AdHoc connection modes
	ENABLE_P2P_PRINTS	not supported
	ENABLE_WPS_PRINTS	This is macro ENABLE_WPS_PRINTS.
	WF_WPS_PIN_LENGTH	WPS PIN Length
	DRV_WIFI_P2P_ERROR_CODES	Selection of different codes during a P2P connection.

	DRV_WIFI_P2P_STATES	Selection of different states during a P2P connection.
	DRV_WIFI_WPS_ERROR_CONFIG_CODES	Selection of different codes when a WPS connection fails.
	DRV_WIFI_WPS_STATE_CODES	Selection of different codes when a Extensible Authentication Protocol is used.
	DRV_GFX_SSD1926_COMMAND	Structure for the commands in the driver queue.
	DRV_WIFI_DEFAULT_WEP_KEY_INDEX	see DRV_WIFI_SecurityWepSet() and DRV_WIFI_WEP_CONTEXT
	DRV_WIFI_SOFTAP_NETWORK_CONTEXT	Contains data pertaining to Wi-Fi SoftAP context
	DRV_WIFI_DEFAULT_SOFTAP_HIDDEN_SSID	Default values for Wi-Fi SoftAP settings

Description

This section describes the Application Programming Interface (API) functions of the MRF24W Wi-Fi Driver.

Refer to each section for a detailed description.

a) Wi-Fi Initialization Functions

DRV_WIFI_Deinitialize Function

Initializes the MRF24WG Wi-Fi driver.

Implementation: Dynamic

File

drv_wifi.h

C

```
bool DRV_WIFI_Deinitialize();
```

Returns

If successful returns true, else false.

Description

This function deinitializes the MRF24WG driver. It also saves the Wi-Fi parameters in non-volatile storage.

Remarks

None

Preconditions

None.

Function

bool DRV_WIFI_Deinitialize(void);

DRV_WIFI_Initialize Function

Initializes the MRF24WG Wi-Fi driver.

Implementation: Dynamic

File

drv_wifi.h

C

```
bool DRV_WIFI_Initialize(void* pNetIf);
```

Returns

If successful returns true, else false.

Description

This function initializes the MRF24WG driver, making it ready for clients to use.

Remarks

This function must be called before any other Wi-Fi routine is called. Currently, this function performs no work, but that may change in the future. The Wi-Fi initialization takes place in a state machine called by MRF24W_MACInit().

Preconditions

None.

Parameters

Parameters	Description
pNetIf	Pointer to network interface

Function

bool DRV_WIFI_Initialize(void* pNetIf);

DRV_WIFI_MRF24W_ISR Function

Wi-Fi driver (MRF24WG specific) interrupt service routine.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_MRF24W_ISR(SYS_MODULE_OBJ index);
```

Returns

None.

Description

This function is Wi-Fi driver (MRF24WG specific) interrupt service routine.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void DRV_WIFI_MRF24W_ISR(SYS_MODULE_OBJ index)

DRV_WIFI_ConnectStateMachine Function

Starts the state machine to connect to an 802.11 network.

Implementation: Dynamic

File

drv_wifi.h

C

```
TCPIP_MAC_RES DRV_WIFI_ConnectStateMachine();
```

Returns

TCP/IP stack MAC result.

Description

This function starts the state machine to connect to an 802.11 network.

Remarks

None.

Preconditions

TCP/IP stack should be initialized.

Function

TCPIP_MAC_RES DRV_WIFI_ConnectStateMachine(void)

b) Wi-Fi Network General Configuration Functions

DRV_WIFI_ChannelListSet Function

Sets the channel list.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ChannelListSet(uint8_t * p_channelList, uint8_t numChannels);
```

Returns

None

Description

This function sets the channel list that the MRF24WG will use when scanning or connecting.

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t channelList[1, 6, 11];

DRV_WIFI_ChannelListSet(channelList, sizeof(channelList));
```

Parameters

Parameters	Description
p_channelList	list of channels
numChannels	number of channels in list; if set to 0, then MRF24WG will set its channel list to all valid channels in its regional domain.

Function

void DRV_WIFI_ChannelListSet(uint8_t *p_channelList, uint8_t numChannels);

DRV_WIFI_NetworkTypeSet Function

Sets the Wi-Fi network type.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_NetworkTypeSet(uint8_t networkType);
```

Returns

None

Description

This function selects the Wi-Fi network type.

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_NetworkTypeSet(DRV_WIFI_NETWORK_TYPE_INFRASTRUCTURE);
```

Parameters

Parameters	Description
networkType	One of the following: DRV_WIFI_NETWORK_TYPE_INFRASTRUCTURE DRV_WIFI_NETWORK_TYPE_ADHOC
DRV_WIFI_NETWORK_TYPE_P2P	not supported DRV_WIFI_NETWORK_TYPE_SOFT_AP

Function

void DRV_WIFI_NetworkTypeSet(uint8_t networkType);

DRV_WIFI_ReconnectModeSet Function

Sets the Wi-Fi reconnection mode.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ReconnectModeSet(uint8_t retryCount, uint8_t deauthAction, uint8_t beaconTimeout,
uint8_t beaconTimeoutAction);
```

Returns

None

Description

This function controls how the MRF24WG behaves when an existing Wi-Fi connection is lost. The MRF24WG can lose an existing connection in one of two ways: 1) Beacon timeout 2) Deauthorization received from AP

There are two options with respect to regaining a lost Wi-Fi connection: 1) MRF24WG informs the host that the connection was temporarily lost and then the MRF24WG retries N times (or forever) to regain the connection. 2) MRF24WG simply informs the host application that the connection is lost, and it is up to the host to regain the connection via the API.

Remarks

The retryCount parameter also applies when initially connecting. That is, the retryCount tells the MRF24WG how many time to try to connect to a Wi-Fi network before giving up and generating the DRV_WIFI_EVENT_CONNECTION_FAILED event.

'retryCount' field	Description
0	Do not try to regain a connection (simply report event to host)
1:254	Number of times MRF24WG should try to regain the connection
255	MRF24WG will retry forever (do not use for AdHoc connections)

'deauthAction' field	Description
DRV_WIFI_DO_NOT_ATTEMPT_TO_RECONNECT	Do not attempt to reconnect after a deauth
DRV_WIFI_ATTEMPT_TO_RECONNECT	Attempt to reconnect after a deauth

'beaconTimeout' field	Description
0	MRF24WG will not monitor the beacon timeout condition
1:255	Number of missed beacons before designating the connection as lost.

'beaconTimeoutAction' field	Description
DRV_WIFI_DO_NOT_ATTEMPT_TO_RECONNECT	Do not attempt to reconnect after a beacon timeout
DRV_WIFI_ATTEMPT_TO_RECONNECT	Attempt to reconnect after a beacon timeout

Preconditions

Wi-Fi initialization must be complete.

Example

```
// Example 1: MRF24WG should retry forever if either a deauth or beacon
// timeout occurs (beacon timeout is 3 beacon periods).
DRV_WIFI_ReconnectModeSet(WF_RETRY_FOREVER,
                          WF_ATTEMPT_TO_RECONNECT,
                          3,
                          WF_ATTEMPT_TO_RECONNECT);

// Example 2: MRF24WG should not do any connection retries and only report
// deauthorization events to the host.
DRV_WIFI_ReconnectModeSet(0,
                          WF_DO_NOT_ATTEMPT_TO_RECONNECT,
                          0,
                          WF_DO_NOT_ATTEMPT_TO_RECONNECT);

// Example 3: MRF24WG should not do any connection retries, but report deauthorization
// and beacon timeout events to host.  Beacon timeout should be 5 beacon periods.
DRV_WIFI_ReconnectModeSet(0,
                          WF_DO_NOT_ATTEMPT_TO_RECONNECT,
                          5,
                          WF_DO_NOT_ATTEMPT_TO_RECONNECT);

// Example 4: MRF24WG should ignore beacon timeouts, but attempt to
// reconnect 3 times if a deauthorization occurs.
DRV_WIFI_ReconnectModeSet(3,
                          WF_ATTEMPT_TO_RECONNECT,
                          0,
                          WF_DO_NOT_ATTEMPT_TO_RECONNECT);
```

Parameters

Parameters	Description
retryCount	Number of times the MRF24WG should try to regain the connection (see description)
deauthAction	In the event of a deauthorization from the AP, the action the MRF24WG should take (see description)
beaconTimeout	Number of missed beacons before the MRF24WG designates the connection as lost (see description)
beaconTimeoutAction	In the event of a beacon timeout, the action the MRF24WG should take (see description)

Function

void DRV_WIFI_ReconnectModeSet(uint8_t retryCount, uint8_t deauthAction,

uint8_t beaconTimeout, uint8_t beaconTimeoutAction);

DRV_WIFI_SsidSet Function

Sets the SSID.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SsidSet(uint8_t * p_ssid, uint8_t ssidLength);
```

Returns

None

Description

Sets the SSID and SSID Length. Note that an Access Point can have either a visible or hidden SSID. If an Access Point uses a hidden SSID then an active scan must be used.

Remarks

Do not include a string terminator in the SSID length. SSIDs are case-sensitive. SSID length must be less than or equal to DRV_WIFI_MAX_SSID_LENGTH.

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t ssid[] = "MySSIDName";
uint8_t ssidLength = strlen(ssid);

DRV_WIFI_SsidSet(ssid, &ssidLength);
```

Parameters

Parameters	Description
p_ssid	Pointer to SSID buffer
ssidLength	number of bytes in SSID

Function

void DRV_WIFI_SsidSet(uint8_t *p_ssid, uint8_t ssidLength);

DRV_WIFI_AdhocContextSet Function

Sets the AdHoc context.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_AdhocContextSet(DRV_WIFI_ADHOC_NETWORK_CONTEXT * p_context);
```

Returns

None

Description

This function sets the AdHoc context. It is only applicable when the DRV_WIFI_NETWORK_TYPE_ADHOC has been selected in DRV_WIFI_NetworkTypeSet().

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_ADHOC_NETWORK_CONTEXT adHocContext;

adHocContext.mode         = DRV_WIFI_ADHOC_CONNECT_THEN_START;
adHocContext.hiddenSsid   = false;
adHocContext.beaconPeriod = DRV_WIFI_DEFAULT_ADHOC_BEACON_PERIOD;

DRV_WIFI_AdhocContextSet(&adHocContext);
```

Parameters

Parameters	Description
p_context	pointer to AdHoc context data; see definition for the DRV_WIFI_ADHOC_NETWORK_CONTEXT structure.

Function

void DRV_WIFI_AdhocContextSet(DRV_WIFI_ADHOC_NETWORK_CONTEXT *p_context);

c) Wi-Fi Network Security Configuration Functions

DRV_WIFI_SetPSK Function

Sets the binary WPA PSK code in WPS.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SetPSK(uint8_t * p_psk);
```

Returns

None.

Description

This function is used in conjunction with DRV_WIFI_YieldPassphraseToHost(). It sends the binary key to the MRF24WG after the host has converted an ASCII passphrase to a binary key.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_YieldPassphraseToHost(&info);
```

Parameters

Parameters	Description
p_psk	pointer to the binary key

Function

void DRV_WIFI_SetPSK(uint8_t *p_psk);

DRV_WIFI_SecurityOpenSet Function

Sets Wi-Fi security to open (no security).

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SecurityOpenSet();
```

Returns

None

Description

This function sets the Wi-Fi security to open. One can only connect to an AP that is running in open mode.

Remarks

None

Preconditions

Wi-Fi initialization must be complete. Must be in an unconnected state.

Example

```
    DRV_WIFI_SecurityOpenSet();
```

Function

void DRV_WIFI_SecurityOpenSet(void);

DRV_WIFI_SecurityWepSet Function

Sets Wi-Fi security to use WEP.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SecurityWepSet(DRV_WIFI_WEP_CONTEXT* p_context);
```

Returns

None

Description

This function sets the Wi-Fi security to WEP. One can only connect to an AP that is running the same WEP mode.

Remarks

None

Preconditions

Wi-Fi initialization must be complete. Must be in an unconnected state.

Example

```
DRV_WIFI_WEP_CONTEXT context;

context.wepSecurityType = DRV_WIFI_SECURITY_WEP_40;
context.wepKey[] = {0x5a, 0xfb, 0x6c, 0x8e, 0x77,
                    0xc1, 0x04, 0x49, 0xfd, 0x4e,
                    0x43, 0x18, 0x2b, 0x33, 0x88,
                    0xb0, 0x73, 0x69, 0xf4, 0x78};

context.wepKeyLength = 20;
context.wepKeyType = DRV_WIFI_SECURITY_WEP_OPENKEY;
DRV_WIFI_SecurityOpenSet(&context);
```

Parameters

Parameters	Description
p_context	desired WEP context. See DRV_WIFI_WEP_CONTEXT structure.

Function

void DRV_WIFI_SecurityWepSet(DRV_WIFI_WEP_CONTEXT* p_context);

DRV_WIFI_SecurityWpaSet Function

Sets Wi-Fi security to use WPA or WPA2.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SecurityWpaSet(DRV_WIFI_WPA_CONTEXT* p_context);
```

Returns

None

Description

This function sets the Wi-Fi security to WPA or WPA2. One can only connect to an AP that is running the same WPA mode.

Remarks

None

Preconditions

Wi-Fi initialization must be complete. Must be in an unconnected state.

Example

```
DRV_WIFI_WPA_CONTEXT context;

context.wpaSecurityType  = DRV_WIFI_SECURITY_WPA_AUTO_WITH_PASS_PHRASE
context.keyInfo.key[]    = "MySecretWPA2PassPhrase";
context.keyInfo.keyLenth = strlen(context.keyInfo.key);
DRV_WIFI_SecurityWpaSet(&context);
```

Parameters

Parameters	Description
p_context	desired WPA context. See DRV_WIFI_WPA_CONTEXT structure.

Function

DRV_WIFI_SecurityWpaSet(DRV_WIFI_WPA_CONTEXT* p_context);

DRV_WIFI_SecurityWpsSet Function

Sets Wi-Fi security to use WPS.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SecurityWpsSet(DRV_WIFI_WPS_CONTEXT * p_context);
```

Returns

None

Description

This function sets the Wi-Fi security to WPS. One can only connect to an AP that supports WPS.

Remarks

None

Preconditions

Wi-Fi initialization must be complete. Must be in an unconnected state.

Example

```
DRV_WIFI_WPS_CONTEXT context;
uint8_t wpsPin[8] = {1, 2, 3, 9, 0, 2, 1, 2};

context.wpsSecurityType = DRV_WIFI_SECURITY_WPS_PUSH_BUTTON;
memcpy(context.wpsPin, wpsPin, sizeof(wpsPin));
context.wpsPinLength = 8;
DRV_WIFI_SecurityWpsSet(&context);
```

Parameters

Parameters	Description
p_context	desired WPA context. See DRV_WIFI_WPS_CONTEXT structure.

Function

void DRV_WIFI_SecurityWpsSet(DRV_WIFI_WPS_CONTEXT *p_context);

d) Wi-Fi Network Power Configuration Functions

DRV_WIFI_HibernateEnable Function

Puts the MRF24WG into hibernate mode.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_HibernateEnable();
```

Returns

None.

Description

Enables Hibernate mode on the MRF24W, which effectively turns off the device for maximum power savings.

MRF24W state is not maintained when it transitions to hibernate mode.

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_HibernateEnable();
```

Function

void DRV_WIFI_HibernateEnable(void)

DRV_WIFI_PsPollDisable Function

Disables PS-Poll mode.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_PsPollDisable();
```

Returns

None.

Description

Disables PS Poll mode. The MRF24W will stay active and not go to sleep.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_PsPollDisable(&context);
```

Function

void DRV_WIFI_PsPollDisable(void)

DRV_WIFI_PsPollEnable Function

Enables PS Poll mode.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_PsPollEnable(DRV_WIFI_PS_POLL_CONTEXT * p_context);
```

Returns

None.

Description

Enables PS Poll mode. PS-Poll (Power-Save Poll) is a mode allowing for longer battery life. The MRF24W coordinates with the Access Point to go to sleep and wake up at periodic intervals to check for data messages, which the Access Point will buffer. The listenInterval in the Connection Algorithm defines the sleep interval. By default, PS-Poll mode is disabled.

When PS Poll is enabled, the Wi-Fi Host Driver will automatically force the MRF24W to wake up each time the Host sends TX data or a control message to the MRF24W. When the Host message transaction is complete the MRF24W driver will automatically re-enable PS Poll mode.

When the application is likely to experience a high volume of data traffic then PS-Poll mode should be disabled for two reasons:

1. No power savings will be realized in the presence of heavy data traffic.
2. Performance will be impacted adversely as the Wi-Fi Host Driver continually activates and deactivates PS-Poll mode via SPI messages.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_PS_POLL_CONTEXT context;

context.listenInterval = DRV_WIFI_DEFAULT_PS_LISTEN_INTERVAL;
context.dtimInterval   = DRV_WIFI_DEFAULT_PS_DTIM_INTERVAL;
context.useDtim        = DRV_WIFI_DEFAULT_PS_DTIM_ENABLED;

DRV_WIFI_PsPollEnable(&context);
```

Parameters

Parameters	Description
p_context	Pointer to ps poll context. See DRV_WIFI_PS_POLL_CONTEXT structure.

Function

void DRV_WIFI_PsPollEnable(DRV_WIFI_PS_POLL_CONTEXT *p_context);

e) Wi-Fi Network Multicast Filter Configuration Functions

DRV_WIFI_HWMulticastFilterSet Function

Sets a multicast address filter using one of the two hardware multicast filters.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_HWMulticastFilterSet(uint8_t multicastFilterId, uint8_t multicastAddress[6]);
```

Returns

None.

Description

This function allows the application to configure up to two hardware Multicast Address Filters on the MRF24W. If two active multicast filters are set up they are ORed together - the MRF24W will receive and pass to the Host CPU received packets from either multicast address. The allowable values for the multicast filter are:

- DRV_WIFI_MULTICAST_FILTER_1
- DRV_WIFI_MULTICAST_FILTER_2

By default, both Multicast Filters are inactive.

Remarks

Cannot mix hardware and software multicast filters.

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t multicastFilterId = DRV_WIFI_MULTICAST_FILTER_1;
uint8_t filterAddress[6] = {0x00 0x01, 0x5e, 0x11, 0x22, 0x33};

DRV_WIFI_HWMulticastFilterSet(multicastFilterId, filterAddress);
```

Parameters

Parameters	Description
multicastFilterId	DRV_WIFI_MULTICAST_FILTER_1 or DRV_WIFI_MULTICAST_FILTER_2
multicastAddress	6 byte address (all 0xFF will inactivate the filter)

Function

void DRV_WIFI_HWMulticastFilterSet(uint8_t multicastFilterId,

uint8_t multicastAddress[6])

DRV_WIFI_SWMulticastFilterSet Function

Sets a multicast address filter using one of the software multicast filters.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SWMulticastFilterSet(DRV_WIFI_SWMULTICAST_CONFIG * p_config);
```

Returns

None.

Description

This function allows the application to configure up to two Multicast Address Filters on the MRF24W. If two active multicast filters are set up they are ORed together - the MRF24W will receive and pass to the Host CPU received packets from either multicast address. The allowable values in p_config are:

filterId -- DRV_WIFI_MULTICAST_FILTER_1 through DRV_WIFI_MULTICAST_FILTER_16

action -- DRV_WIFI_MULTICAST_DISABLE_ALL (default) The Multicast Filter discards all received multicast messages - they will not be forwarded to the Host PIC. The remaining fields in this structure are ignored.

DRV_WIFI_MULTICAST_ENABLE_ALL The Multicast Filter forwards all received multicast messages to the Host PIC. The remaining fields in this structure are ignored.

DRV_WIFI_MULTICAST_USE_FILTERS The MAC filter will be used and the remaining fields in this structure configure which Multicast messages are forwarded to the Host PIC.

macBytes -- Array containing the MAC address to filter on (using the destination address of each incoming 802.11 frame). Specific bytes with the MAC address can be designated as "don't care" bytes. See macBitMask. This field in only used if action = DRV_WIFI_MULTICAST_USE_FILTERS.

macBitMask -- A byte where bits 5:0 correspond to macBytes[5:0]. If the bit is zero then the corresponding MAC byte must be an exact match for the frame to be forwarded to the Host PIC. If the bit is one then the corresponding MAC byte is a "don't care" and not used in the Multicast filtering process. This field in only used if action = DRV_WIFI_MULTICAST_USE_FILTERS.

Remarks

Cannot mix hardware and software multicast filters..

Preconditions

Wi-Fi initialization must be complete. DRV_WIFI_SWMultiCastFilterEnable() must have been called previously.

Example

```
DRV_WIFI_SWMULTICAST_CONFIG config;
uint8_t macMask[] = {01, 00, 5e, ff, ff, ff};  // (0xff are the don't care bytes)

// configure software multicast filter 1 to filter multicast addresses that
// start with 01:00:5e
config.action = DRV_WIFI_MULTICAST_USE_FILTERS;
config->filterId = DRV_WIFI_MULTICAST_FILTER_1;
memcpy(config->macBytes, macMask, 6);
config->macBitMask = 0x38;            // bits 5:3 = 1 (don't care on bytes 3,4,5)
                                      // bits 2:0 = 0 (exact match required on bytes 0, 1, 2)
```

Function

void DRV_WIFI_SWMulticastFilterSet(DRV_WIFI_SWMULTICAST_CONFIG *p_config);

f) Wi-Fi Network Connection Functions

DRV_WIFI_Connect Function

Directs the MRF24WG to connect to a Wi-Fi network.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_Connect();
```

Returns

None

Description

This function causes the MRF24WG to connect to a Wi-Fi network. Upon connection, or a failure to connect, an event will be generated.

Remarks

None

Preconditions

Wi-Fi initialization must be complete and relevant connection parameters must have been set.

Example

```
    DRV_WIFI_Connect();
```

Function

void DRV_WIFI_Connect(void);

DRV_WIFI_Disconnect Function

Directs the MRF24WG to disconnect from a Wi-Fi network.

Implementation: Dynamic

File

drv_wifi.h

C

```
uint16_t DRV_WIFI_Disconnect();
```

Returns

DRV_WIFI_SUCCESS or DRV_WIFI_ERROR_DISCONNECT_FAILED

Description

This function causes the MRF24WG to disconnect from a Wi-Fi network. No event is generated when a connection is terminated via the function call.

Remarks

None

Preconditions

Wi-Fi initialization must be complete and a connection must be in progress.

Example

```
DRV_WIFI_Disconnect();
```

Function

uint16_t DRV_WIFI_Disconnect(void);

g) Wi-Fi Network Gratuitous ARP Configuration Functions

DRV_WIFI_GratuitousArpStart Function

Starts a periodic gratuitous ARP response.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_GratuitousArpStart(uint8_t period);
```

Returns

None.

Description

This function starts a gratuitous ARP response to be periodically transmitted.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete. Connection process must be complete.

Example

```
DRV_WIFI_GratuitousArpStart(10); // begin sending gratuitous ARPs every
                                 // 10 seconds.
```

Parameters

Parameters	Description
period	period between gratuitous ARP, in seconds

Function

void DRV_WIFI_GratuitousArpStart(uint8_t period)

DRV_WIFI_GratuitousArpStop Function

Stops a periodic gratuitous ARP.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_GratuitousArpStop();
```

Returns

None.

Description

This function stops a gratuitous ARP.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_GratuitousArpStop();
```

Function

void DRV_WIFI_GratuitousArpStop(void)

h) Wi-Fi Network Configuration - Other Functions

DRV_WIFI_BssidSet Function

Sets the Basic Service Set Identifier (BSSID).

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_BssidSet(uint8_t * p_bssid);
```

Returns

None

Description

This sets 6 byte (48-bit) MAC address of the Access Point that is being scanned for. It is optional to use this. Where it is useful is if there are two APs with the same ID; the BSSID is used to connect to the specified AP. This setting can be used in lieu of the SSID. Set each byte to 0xFF (default) if the BSSID is not being used. Not typically needed.

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t bssid[6];

bssid[0] = 0x00;
bssid[1] = 0xe8;
bssid[2] = 0xc0;
bssid[3] = 0x11;
bssid[4] = 0x22;
bssid[5] = 0x33;

DRV_WIFI_BssidSet(bssid);
```

Parameters

Parameters	Description
p_context	pointer to BSSID

Function

void DRV_WIFI_BssidSet(uint8_t *p_bssid);

DRV_WIFI_SetLinkDownThreshold Function

Sets number of consecutive Wi-Fi TX failures before link is considered down.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SetLinkDownThreshold(uint8_t threshold);
```

Returns

None.

Description

This function allows the application to set the number of MRF24W consecutive TX failures before the connection failure event (DRV_WIFI_LINK_LOST) is reported to the host application.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete

Example

```
DRV_WIFI_SetLinkDownThreshold(0); // disable link down threshold
```

Parameters

Parameters	Description
threshold	0: disabled (default)
1-255	number of consecutive TX failures before connection failure event is reported

Function

void DRV_WIFI_SetLinkDownThreshold(uint8_t threshold)

DRV_WIFI_SWMultiCastFilterEnable Function

Forces the MRF24WG to use software multicast filters instead of hardware multicast filters.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SWMultiCastFilterEnable();
```

Returns

None.

Description

This function allows the application to configure up to 16 software multicast address Filters on the MRF24WG0MA/B.

Remarks

Cannot mix hardware and software multicast filters..

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_SWMultiCastFilterEnable();
```

Function

void DRV_WIFI_SWMultiCastFilterEnable(void)

DRV_WIFI_MacAddressSet Function

Uses a different MAC address for the MRF24W.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_MacAddressSet(uint8_t * p_mac);
```

Returns

None.

Description

Directs the MRF24W to use the input MAC address instead of its factory-default MAC address. This function does not overwrite the factory default, which is in Flash memory.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete. Cannot be called when the MRF24W is in a connected state.

Example

```
uint8_t mac[6] = {0x00, 0x1e, 0xc0, 0x11, 0x22, 0x33};

DRV_WIFI_MacAddressSet(mac);
```

Parameters

Parameters	Description
p_mac	Pointer to 6 byte MAC that will be sent to MRF24WG

Function

void DRV_WIFI_MacAddressSet(uint8_t *p_mac)

DRV_WIFI_RtsThresholdSet Function

Sets the RTS Threshold.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_RtsThresholdSet(uint16_t rtsThreshold);
```

Returns

None.

Description

Sets the RTS/CTS packet size threshold for when RTS/CTS frame will be sent. The default is 2347 bytes - the maximum for 802.11. It is recommended that the user leave the default at 2347 until they understand the performance and power ramifications of setting it smaller. Valid values are from 0 to DRV_WIFI_RTS_THRESHOLD_MAX (2347).

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_RtsThresholdSet(DRV_WIFI_RTS_THRESHOLD_MAX);
```

Parameters

Parameters	Description
rtsThreshold	Value of the packet size threshold

Function

void DRV_WIFI_RtsThresholdSet(uint16_t rtsThreshold)

DRV_WIFI_ScanContextSet Function

Sets the Wi-Fi scan context.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ScanContextSet(DRV_WIFI_SCAN_CONTEXT * p_context);
```

Returns

None.

Description

This function sets the Wi-Fi scan context. The MRF24WG defaults are fine for most applications, but they can be changed by this function.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_SCAN_CONTEXT context;

context.scantype = DRV_WIFI_ACTIVE_SCAN;
context.scanCount      = 1;
context.minChannelTime = 200;    // ms
context.maxChannelTime = 400;    // ms
context.probeDelay     = 20;     // uS

DRV_WIFI_ScanContextSet(&context);
```

Parameters

Parameters	Description
p_context	Pointer to scan context. See DRV_WIFI_SCAN_CONTEXT structure.

Function

void DRV_WIFI_ScanContextSet(DRV_WIFI_SCAN_CONTEXT *p_context)

DRV_WIFI_TxModeSet Function

Configures 802.11 TX mode.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_TxModeSet(uint8_t mode);
```

Returns

None.

Description

This function sets the MRF24WG TX mode.

mode	Description
DRV_WIFI_TXMODE_G_RATES	Use all 802.11g rates (default)
DRV_WIFI_TXMODE_B_RATES	Use only 802.11b rates
DRV_WIFI_TXMODE_LEGACY_RATES	Use only 1 and 2 mbps rates

Remarks

None.

Preconditions

Wi-Fi initialization must be complete

Example

```
DRV_WIFI_TxModeSet(DRV_WIFI_TXMODE_G_RATES);
```

Parameters

Parameters	Description
mode	TX mode value (see description)

Function

void DRV_WIFI_TxModeSet(uint8_t mode)

© 2013-2015 Microchip Technology Inc.

DRV_WIFI_RssiSet Function

Sets RSSI restrictions when connecting.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_RssiSet(uint8_t rssi);
```

Returns

None

Description

This setting is only used if: 1) Neither an SSID or BSSID has been configured or 2) An SSID is defined and multiple APs are discovered with the same SSID

rssi	Description
0	Connect to first network found
1-254	Only connect to network if the RSSI is greater or equal to this value
255	Connect to the highest RSSI found (default)

Remarks

Rarely needed

Preconditions

Wi-Fi initialization must be complete

Example

```
DRV_WIFI_RssiSet(255);
```

Parameters

Parameters	Description
rssi	See description

Function

void DRV_WIFI_RssiSet(uint8_t rssi);

DRV_WIFI_EasyConfigTask_RtosTask Function

Implements Wi-Fi driver easy configuration RTOS task.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_EasyConfigTask_RtosTask(void * p_arg);
```

Returns

None.

Description

This function implements Wi-Fi driver easy configuration RTOS task.

Remarks

None.

Preconditions

TCP/IP stack should be initialized. DRV_WIFI_RTOS_TaskInit() must has been called previously.

Function

void DRV_WIFI_EasyConfigTask_RtosTask(void *p_arg)

DRV_WIFI_InitStateMachine_RtosTask Function

Implements Wi-Fi driver initialization RTOS task.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_InitStateMachine_RtosTask(void * p_arg);
```

Returns

None.

Description

This function implements Wi-Fi driver initialization RTOS task.

Remarks

None.

Preconditions

TCP/IP stack should be initialized. DRV_WIFI_RTOS_TaskInit() must has been called previously.

Function

void DRV_WIFI_InitStateMachine_RtosTask(void *p_arg)

DRV_WIFI_ISR_RtosTask Function

Implements Wi-Fi driver ISR RTOS task.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ISR_RtosTask(void * p_arg);
```

Returns

None.

Description

This function implements Wi-Fi driver ISR RTOS task.

Remarks

None.

Preconditions

TCP/IP stack should be initialized. DRV_WIFI_RTOS_TaskInit() must has been called previously.

Function

void DRV_WIFI_ISR_RtosTask(void *p_arg)

DRV_WIFI_ISR_SemUnlock Function

Unlocks semaphore in Wi-Fi RTOS ISR.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ISR_SemUnlock();
```

Returns

None.

Description

This function unlocks semaphore in Wi-Fi RTOS ISR.

Remarks

None.

Preconditions

TCP/IP stack should be initialized. DRV_WIFI_RTOS_TaskInit() must has been called previously.
DRV_WIFI_ISR_SemLock() must has been called previously.

Function

void DRV_WIFI_ISR_SemUnlock(void)

DRV_WIFI_MACProcess_RtosTask Function

Implements Wi-Fi driver MAC process RTOS task.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_MACProcess_RtosTask(void * p_arg);
```

Returns

None.

Description

This function implements Wi-Fi driver MAC process RTOS task.

Remarks

None.

Preconditions

TCP/IP stack should be initialized. DRV_WIFI_RTOS_TaskInit() must has been called previously.

Function

void DRV_WIFI_MACProcess_RtosTask(void *p_arg)

DRV_WIFI_TASK_MUTEX_Lock Function

Locks MUTEX in Wi-Fi RTOS task when necessary.

Implementation: Dynamic

File

drv_wifi.h

C

```
bool DRV_WIFI_TASK_MUTEX_Lock();
```

Returns

true or false

Description

This function locks MUTEX in Wi-Fi RTOS task when necessary.

Remarks

None.

Preconditions

TCP/IP stack should be initialized. DRV_WIFI_RTOS_TaskInit() must has been called previously.

Function

bool DRV_WIFI_TASK_MUTEX_Lock(void)

DRV_WIFI_TASK_MUTEX_Unlock Function

Unlocks MUTEX in Wi-Fi RTOS task.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_TASK_MUTEX_Unlock();
```

Returns

None.

Description

This function unlocks MUTEX in Wi-Fi RTOS task.

Remarks

None.

Preconditions

TCP/IP stack should be initialized. DRV_WIFI_RTOS_TaskInit() must has been called previously.
DRV_WIFI_TASK_MUTEX_Lock() must has been called previously.

Function

void DRV_WIFI_TASK_MUTEX_Unlock(void)

DRV_WIFI_RSSI_Cache_FromRxDataRead Function

Caches RSSI value from Rx data packet.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_RSSI_Cache_FromRxDataRead(uint16_t rssi);
```

Returns

None.

Description

This function caches RSSI value from Rx data packet.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
rssi	RSSI value read from Rx data packet

Function

void DRV_WIFI_RSSI_Cache_FromRxDataRead(uint16_t rssi)

DRV_WIFI_RSSI_Get_FromRxDataRead Function

Reads RSSI value from Rx data packet.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_RSSI_Get_FromRxDataRead(uint16_t * mean, uint16_t * last);
```

Returns

mean - the calculated mean RSSI. last - the total count of RSSI values.

Description

This function reads RSSI value from Rx data packet.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
mean	pointer to where the average mean RSSI value to be stored.
last	pointer to where the total count of RSSI values to be stored.

Function

void DRV_WIFI_RSSI_Get_FromRxDataRead(uint16_t *mean, uint16_t *last)

DRV_WIFI_INT_Handle Function

Wi-Fi driver interrupt handle.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_INT_Handle();
```

Returns

None.

Description

This function is the interrupt handle of Wi-Fi driver.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void DRV_WIFI_INT_Handle(void)

DRV_WIFI_ISR_SemLock Function

Locks semaphore in Wi-Fi RTOS ISR.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ISR_SemLock();
```

Returns

None.

Description

This function locks semaphore in Wi-Fi RTOS ISR.

Remarks

None.

Preconditions

TCP/IP stack should be initialized. DRV_WIFI_RTOS_TaskInit() must has been called previously.

Function

void DRV_WIFI_ISR_SemLock(void)

DRV_WIFI_RTOS_TaskInit Function

Initializes RTOS tasks for Wi-Fi driver.

Implementation: Dynamic

File

drv_wifi.h

C

```
bool DRV_WIFI_RTOS_TaskInit();
```

Returns

true or false

Description

This function initializes RTOS tasks for Wi-Fi driver.

Remarks

None.

Preconditions

TCP/IP stack should be initialized.

Function

bool DRV_WIFI_RTOS_TaskInit(void)

DRV_WIFI_SpiClose Function

Closes SPI object for Wi-Fi driver.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SpiClose();
```

Returns

None.

Description

This function closes SPI object for Wi-Fi driver.

Remarks

None.

Preconditions

TCP/IP stack should be initialized.

Function

void DRV_WIFI_SpiClose(void)

DRV_WIFI_SpiInit Function

Initializes SPI object for Wi-Fi driver.

Implementation: Dynamic

File

drv_wifi.h

C

```
bool DRV_WIFI_SpiInit();
```

Returns

true or false

Description

This function initializes SPI object for Wi-Fi driver.

Remarks

None.

Preconditions

TCP/IP stack should be initialized.

Function

bool DRV_WIFI_SpiInit(void)

i) Wi-Fi Scanning Functions

DRV_WIFI_Scan Function

Commands the MRF24W to start a scan operation. This will generate the WF_EVENT_SCAN_RESULTS_READY event.

Implementation: Dynamic

File

drv_wifi.h

C

```
uint16_t DRV_WIFI_Scan(bool scanAll);
```

Returns

None.

Description

Directs the MRF24W to initiate a scan operation. The Host Application will be notified that the scan results are ready when it receives the WF_EVENT_SCAN_RESULTS_READY event. The eventInfo field for this event will contain the number of scan results. Once the scan results are ready they can be retrieved with DRV_WIFI_ScanGetResult().

Scan results are retained on the MRF24W until:

1. Calling DRV_WIFI_Scan() again (after scan results returned from previous call).
2. MRF24W reset.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_Scan(true);
```

Parameters

Parameters	Description
scanAll	If false: • If SSID defined then only scan results with that SSID are retained. • If SSID not defined then all scanned SSIDs will be retained • Only scan results from Infrastructure or AdHoc networks are retained • The channel list that is scanned will be determined from the channels passed in via DRV_WIFI_ChannelListSet().
If true	• All scan results are retained (both Infrastructure and AdHoc networks). • All channels within the MRF24W's regional domain will be scanned.

Function

uint16_t DRV_WIFI_Scan(bool scanAll)

DRV_WIFI_ScanGetResult Function

Read selected scan results back from MRF24W.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ScanGetResult(uint8_t listIndex, DRV_WIFI_SCAN_RESULT * p_scanResult);
```

Returns

None.

Description

After a scan has completed this function is used to read one scan result at a time from the MRF24WG.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete. WF_EVENT_SCAN_RESULTS_READY event must have already occurred.

Example

```
DRV_WIFI_SCAN_RESULT scanResult;

DRV_WIFI_ScanGetResult(0, &scanResult); // get first scan result in list
```

Parameters

Parameters	Description
listIndex	Index (0 based list) of the scan entry to retrieve.
p_scanResult	Pointer to where scan result is written. See DRV_WIFI_SCAN_RESULT structure.

Function

void DRV_WIFI_ScanGetResult(uint8_t listIndex, t_wfScanResult *p_scanResult)

DRV_WIFI_ScanContextGet Function

Gets the Wi-Fi scan context.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ScanContextGet(DRV_WIFI_SCAN_CONTEXT * p_context);
```

Returns

None.

Description

This function gets the Wi-Fi scan context.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_SCAN_CONTEXT context;

DRV_WIFI_ScanContextSet(&context);
```

Parameters

Parameters	Description
p_context	Pointer to where scan context will be written. See DRV_WIFI_SCAN_CONTEXT structure.

Function

void DRV_WIFI_ScanContextGet(DRV_WIFI_SCAN_CONTEXT *p_context)

j) Wi-Fi Event Processing Functions

DRV_WIFI_ProcessEvent Function

Processes Wi-Fi event.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ProcessEvent(uint16_t event, uint16_t eventInfo);
```

Returns

None.

Description

This function is called to process a Wi-Fi event.

Remarks

None.

Preconditions

TCPIP stack should be initialized.

Parameters

Parameters	Description
event	event code
eventInfo	additional information about the event. Not all events have associated info, in which case this value will be set to DRV_WIFI_NO_ADDITIONAL_INFO (0xff)

Function

void DRV_WIFI_ProcessEvent(uint16_t event, uint16_t eventInfo)

DRV_WIFI_SoftApEventInfoGet Function

Gets the stored Soft AP event info.

Implementation: Dynamic

File

drv_wifi.h

C

```
DRV_WIFI_MGMT_INDICATE_SOFT_AP_EVENT * DRV_WIFI_SoftApEventInfoGet();
```

Returns

None.

Description

This function retrieves the additional event info after a Soft AP event has occurred.

Remarks

None.

Preconditions

Soft AP event must have occurred

Example

```
DRV_WIFI_MGMT_INDICATE_SOFT_AP_EVENT info;;

DRV_WIFI_WPSCredentialsGet(&info);
```

Function

DRV_WIFI_MGMT_INDICATE_SOFT_AP_EVENT * DRV_WIFI_SoftApEventInfoGet(void);

DRV_WIFI_SoftAPContextSet Function

Sets the SoftAP context.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SoftAPContextSet(DRV_WIFI_SOFTAP_NETWORK_CONTEXT * p_context);
```

Returns

None

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_SOFTAP_NETWORK_CONTEXT SoftAPContext;

//SoftAPContext.mode          = DRV_WIFI_ADHOC_CONNECT_THEN_START;
SoftAPContext.hiddenSsid   = false;
//SoftAPContext.beaconPeriod = DRV_WIFI_DEFAULT_ADHOC_BEACON_PERIOD;

DRV_WIFI_SoftAPContextSet(&SoftAPContext);
```

Parameters

Parameters	Description
p_context	pointer to SoftAP context data; see definition for the DRV_WIFI_SOFTAP_NETWORK_CONTEXT structure.

Function

void DRV_WIFI_SoftAPContextSet(DRV_WIFI_SOFTAP_NETWORK_CONTEXT *p_context);

k) Wi-Fi Data Configuration Functions

DRV_WIFI_ConfigDataErase Function

Erases configuration data from the board EEPROM.

Implementation: Dynamic

File

drv_wifi.h

C

```
bool DRV_WIFI_ConfigDataErase();
```

Returns

None.

Description

This function erases configuration data from the board EEPROM.

Remarks

None.

Preconditions

TCP/IP stack should be initialized.

Function

bool DRV_WIFI_ConfigDataErase(void)

DRV_WIFI_ConfigDataLoad Function

Loads configuration data from the board EEPROM.

Implementation: Dynamic

File

drv_wifi.h

C

```
bool DRV_WIFI_ConfigDataLoad();
```

Returns

None.

Description

This function loads configuration data from the board EEPROM. If not present or corrupted then default values will be used.

Remarks

None.

Preconditions

TCP/IP stack should be initialized.

Function

bool DRV_WIFI_ConfigDataLoad(void)

DRV_WIFI_ConfigDataPrint Function

Outputs to console the configuration data from the board EEPROM.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ConfigDataPrint();
```

Returns

None.

Description

This function outputs configuration data from the board EEPROM.

Remarks

None.

Preconditions

TCP/IP stack should be initialized.

Function

void DRV_WIFI_ConfigDataPrint(void)

© 2013-2015 Microchip Technology Inc.

DRV_WIFI_ConfigDataSave Function

Save configuration data to the board EEPROM.

Implementation: Dynamic

File

drv_wifi.h

C

```
bool DRV_WIFI_ConfigDataSave();
```

Returns

None.

Description

This function saves configuration data to the board EEPROM.

Remarks

None.

Preconditions

TCP/IP stack should be initialized.

Function

bool DRV_WIFI_ConfigDataSave(void)

I) Wi-Fi Status Functions

DRV_WIFI_BssidGet Function

Gets the BSSID set in DRV_WIFI_BssidSet().

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_BssidGet(uint8_t * p_bssid);
```

Returns

None

Description

Retrieves the BSSID set in the previous call to DRV_WIFI_BssidSet().

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t bssid[6];

DRV_WIFI_BssidGet(bssid);
```

Parameters

Parameters	Description
p_context	pointer to where BSSID will be written.

Function

void DRV_WIFI_BssidGet(uint8_t *p_bssid);

DRV_WIFI_ChannelListGet Function

Gets the channel list.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ChannelListGet(uint8_t * p_channelList, uint8_t * p_numChannels);
```

Returns

None

Description

This function gets the current channel list.

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t channelList[DRV_WIFI_MAX_CHANNEL_LIST_LENGTH];
uint8_t numChannels;

DRV_WIFI_ChannelListGet(channelList, &numChannels);
```

Parameters

Parameters	Description
p_channelList	pointer to where channel list will be written
numChannels	pointer to where number of channels in the list will be written.

Function

void RV_WIFI_ChannelListGet(uint8_t *p_channelList, uint8_t *p_numChannels);

DRV_WIFI_ConnectContextGet Function

Gets the current Wi-Fi connection context.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ConnectContextGet(DRV_WIFI_CONNECTION_CONTEXT * p_ctx);
```

Returns

None

Description

This function gets the current Wi-Fi connection context.

Remarks

None

Preconditions

Wi-Fi initialization must be complete

Example

```
DRV_WIFI_CONNECTION_CONTEXT ctx;

DRV_WIFI_ConnectContextGet(&ctx);
```

Parameters

Parameters	Description
p_ctx	pointer to where connection context is written. See DRV_WIFI_CONNECTION_CONTEXT structure.

Function

void DRV_WIFI_ConnectContextGet(DRV_WIFI_CONNECTION_CONTEXT *p_ctx);

DRV_WIFI_ConnectionStateGet Function

Gets the current Wi-Fi connection state.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ConnectionStateGet(uint8_t * p_state);
```

Returns

None

Description

This function gets the current Wi-Fi connection state.

*p_state	Description
DRV_WIFI_CSTATE_NOT_CONNECTED	No Wi-Fi connection exists
DRV_WIFI_CSTATE_CONNECTION_IN_PROGRESS	Wi-Fi connection in progress
DRV_WIFI_CSTATE_CONNECTED_INFRASTRUCTURE	Wi-Fi connected in infrastructure mode
DRV_WIFI_CSTATE_CONNECTED_ADHOC	Wi-Fi connected in adhoc mode
DRV_WIFI_CSTATE_RECONNECTION_IN_PROGRESS	Wi-Fi in process of reconnecting
DRV_WIFI_CSTATE_CONNECTION_PERMANENTLY_LOST	Wi-Fi connection permanently lost

Remarks

None

Preconditions

Wi-Fi initialization must be complete

Example

```
uint8_t state;

DRV_WIFI_ConnectionStateGet(&state);
```

Parameters

Parameters	Description
p_state	pointer to where state is written (see description)

Function

void DRV_WIFI_ConnectionStateGet(uint8_t *p_state);

DRV_WIFI_DeviceInfoGet Function

Retrieves MRF24WG device information.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_DeviceInfoGet(DRV_WIFI_DEVICE_INFO * p_deviceInfo);
```

Returns

None.

Description

This function retrieves MRF24WG device information. See DRV_WIFI_DEVICE_INFO structure.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_DEVICE_INFO    info;

DRV_WIFI_DeviceInfoGet(&info);
```

Parameters

Parameters	Description
p_deviceInfo	Pointer where device info will be written

Function

void DRV_WIFI_DeviceInfoGet(DRV_WIFI_DEVICE_INFO *p_deviceInfo)

DRV_WIFI_HWMulticastFilterGet Function

Gets a multicast address filter from one of the two multicast filters.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_HWMulticastFilterGet(uint8_t multicastFilterId, uint8_t multicastAddress[6]);
```

Returns

None.

Description

Gets the current state of the specified Multicast Filter.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t filterAddress[6];

DRV_WIFI_HWMulticastFilterGet(DRV_WIFI_MULTICAST_FILTER_1, filterAddress);
```

Parameters

Parameters	Description
multicastFilterId	DRV_WIFI_MULTICAST_FILTER_1 or DRV_WIFI_MULTICAST_FILTER_2
multicastAddress	pointer to where 6 byte multicast filter is written.

Function

void DRV_WIFI_HWMulticastFilterGet(uint8_t multicastFilterId,

uint8_t multicastAddress[6])

DRV_WIFI_MacAddressGet Function

Retrieves the MRF24WG MAC address.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_MacAddressGet(uint8_t * p_mac);
```

Returns

None.

Description

This function retrieves the MRF24WG MAC address.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t mac[6];

DRV_WIFI_MacAddressGet(mac);
```

Parameters

Parameters	Description
p_mac	Pointer where mac address will be written (must point to a 6 byte buffer)

Function

void DRV_WIFI_MacAddressGet(uint8_t *p_mac)

DRV_WIFI_MacStatsGet Function

Gets MAC statistics.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_MacStatsGet(DRV_WIFI_MAC_STATS * p_macStats);
```

Returns

None.

Description

This function gets the various MAC layer stats as maintained by the MRF24WG.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_MAC_STATS macStats;

DRV_WIFI_MacStatsGet(&macStats);
```

Parameters

Parameters	Description
p_macStats	Pointer to where MAC statistics are written. See DRV_WIFI_MAC_STATS structure.

Function

void DRV_WIFI_MacStatsGet(DRV_WIFI_MAC_STATS *p_macStats)

DRV_WIFI_NetworkTypeGet Function

Gets the Wi-Fi network type.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_NetworkTypeGet(uint8_t * p_networkType);
```

Returns

None

Description

This function gets the Wi-Fi network type.

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t networkType;

DRV_WIFI_NetworkTypeGet(&networkType);
```

Parameters

Parameters	Description
p_networkType	Pointer to where the network type will be written. One of the following: DRV_WIFI_NETWORK_TYPE_INFRASTRUCTURE DRV_WIFI_NETWORK_TYPE_ADHOC
DRV_WIFI_NETWORK_TYPE_P2P	not supported DRV_WIFI_NETWORK_TYPE_SOFT_AP

Function

void DRV_WIFI_NetworkTypeGet(uint8_t *p_networkType);

DRV_WIFI_PowerSaveStateGet Function

Gets the current power-saving state.

File

drv_wifi.h

C

```
void DRV_WIFI_PowerSaveStateGet(uint8_t * p_powerSaveState);
```

Returns

None.

Description

This function gets the current MRF24WG power-saving state.

powerSaveState	Definition
DRV_WIFI_PS_HIBERNATE	MRF24W in hibernate state
DRV_WIFI_PS_PS_POLL_DTIM_ENABLED	MRF24W in PS-Poll mode with DTIM enabled
DRV_WIFI_PS_PS_POLL_DTIM_DISABLED	MRF24W in PS-Poll mode with DTIM disabled
DRV_WIFI_PS_POLL_OFF	MRF24W is not in any power-saving state

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t state;

DRV_WIFI_PowerSaveStateGet(&state);
```

Parameters

Parameters	Description
p_powerSaveState	Pointer to where power state is written (see description)

Function

void DRV_WIFI_PowerSaveStateGet(uint8_t *p_powerSaveState)

DRV_WIFI_ReconnectModeGet Function

Gets the Wi-Fi reconnection mode.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_ReconnectModeGet(uint8_t * p_retryCount, uint8_t * p_deauthAction, uint8_t *
p_beaconTimeout, uint8_t * p_beaconTimeoutAction);
```

Returns

None

Description

This function gets the reconnection mode parameters set in DRV_WIFI_ReconnectModeGet().

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t retryCount, deauthAction, beaconTimeout, beaconTimeoutAction;

DRV_WIFI_ReconnectModeGet(&retryCount,
                          &deauthAction,
                          &beaconTimeout,
                          &beaconTimeoutAction);
```

Parameters

Parameters	Description
p_retryCount	Pointer where retry count is written
p_deauthAction	Pointer where deauthorization action is written
p_beaconTimeout	Pointer where beacon timeout is written
p_beaconTimeoutAction	Pointer where beacon timeout action is written.

Function

void DRV_WIFI_ReconnectModeGet(uint8_t *p_retryCount, uint8_t *p_deauthAction,

uint8_t *p_beaconTimeout, uint8_t *p_beaconTimeoutAction);

DRV_WIFI_RegionalDomainGet Function

Retrieves the MRF24WG Regional domain.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_RegionalDomainGet(uint8_t * p_regionalDomain);
```

Returns

None.

Description

Gets the regional domain on the MRF24W. Values are:

- DRV_WIFI_DOMAIN_FCC
- DRV_WIFI_DOMAIN_ETSI
- DRV_WIFI_DOMAIN_JAPAN
- DRV_WIFI_DOMAIN_OTHER

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t domain;

DRV_WIFI_RegionalDomainGet(&domain);
```

Parameters

Parameters	Description
p_regionalDomain	Pointer where the regional domain value will be written

Function

void DRV_WIFI_RegionalDomainGet(uint8_t *p_regionalDomain)

DRV_WIFI_RtsThresholdGet Function

Gets the RTS Threshold.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_RtsThresholdGet(uint16_t * p_rtsThreshold);
```

Returns

None.

Description

Gets the RTS/CTS packet size threshold.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint16_t threshold;

DRV_WIFI_RtsThresholdGet(&threshold);
```

Parameters

Parameters	Description
p_rtsThreshold	Pointer to where RTS threshold is written

Function

void DRV_WIFI_RtsThresholdGet(uint16_t *p_rtsThreshold)

DRV_WIFI_SecurityGet Function

Gets the current Wi-Fi security setting.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SecurityGet(uint8_t * p_securityType, uint8_t * p_securityKey, uint8_t *
p_securityKeyLength);
```

Returns

None.

Description

This function gets the current Wi-Fi security setting.

'securityType' Field	Key	Length
DRV_WIFI_SECURITY_OPEN	N/A	N/A
DRV_WIFI_SECURITY_WEP_40	binary	4 keys, 5 bytes each (total of 20 bytes)
DRV_WIFI_SECURITY_WEP_104	binary	4 keys, 13 bytes each (total of 52 bytes)
DRV_WIFI_SECURITY_WPA_AUTO_WITH_KEY	binary	32 bytes
DRV_WIFI_SECURITY_WPA_AUTO_WITH_PASS_PHRASE	ascii	8-63 ascii characters

Remarks

If security was initially set with a passphrase that the MRF24WG used to generate a binary key, this function returns the binary key, not the passphrase.

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t securityType;
uint8_t securityKey[DRV_WIFI_MAX_SECURITY_KEY_LENGTH];
uint8_t keyLength;

DRV_WIFI_SecurityGet(&securityType, securityKey, &keyLength);
```

Parameters

Parameters	Description
p_securityType	Value corresponding to the security type desired (see description)
p_securityKey	Binary key or passphrase (not used if security is DRV_WIFI_SECURITY_OPEN)
p_securityKeyLength	Number of bytes in p_securityKey (not used if security is DRV_WIFI_SECURITY_OPEN)

Function

void DRV_WIFI_SecurityGet(uint8_t *p_securityType,

uint8_t *p_securityKey,

uint8_t *p_securityKeyLength)

DRV_WIFI_SsidGet Function

Gets the SSID.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_SsidGet(uint8_t * p_ssid, uint8_t * p_ssidLength);
```

Returns

None

Description

Gets the SSID and SSID Length.

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t ssid[DRV_WIFI_MAX_SSID_LENGTH];
uint8_t ssidLength;

DRV_WIFI_SsidGet(ssid, &ssidLength);
```

Parameters

Parameters	Description
p_ssid	Pointer to buffer where SSID will be written
ssidLength	number of bytes in SSID

Function

void DRV_WIFI_SsidGet(uint8_t *p_ssid, uint8_t *p_ssidLength);

DRV_WIFI_TxModeGet Function

Gets 802.11 TX mode.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_TxModeGet(uint8_t * p_mode);
```

Returns

None.

Description

This function gets the MRF24WG TX mode.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete

Example

```
uint8_t mode;

DRV_WIFI_TxModeGet(&mode);
```

Function

void DRV_WIFI_TxModeGet(uint8_t *p_mode);

DRV_WIFI_WepKeyTypeGet Function

Gets the WEP Key type.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_WepKeyTypeGet(uint8_t * p_wepKeyType);
```

Returns

None.

Description

This function gets the WEP key type:

- DRV_WIFI_SECURITY_WEP_SHAREDKEY
- DRV_WIFI_SECURITY_WEP_OPENKEY

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t wepKeyType;

DRV_WIFI_WepKeyTypeGet(&wepKeyType);
```

Function

void DRV_WIFI_WepKeyTypeGet(uint8_t *p_keyType)

DRV_WIFI_WPSCredentialsGet Function

Gets the WPS credentials.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_WPSCredentialsGet(DRV_WIFI_WPS_CREDENTIAL * p_cred);
```

Returns

None.

Description

This function gets the WPS credentials from the MRF24WG

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_WPS_CREDENTIAL cred;;

DRV_WIFI_WPSCredentialsGet(&cred);
```

Function

void DRV_WIFI_WPSCredentialsGet(DRV_WIFI_WPS_CREDENTIAL *p_cred);

DRV_WIFI_RssiGet Function

Gets RSSI value set in DRV_WIFI_RssiSet().

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_RssiGet(uint8_t * p_rssi);
```

Returns

None

Description

This function retrieves the value set in Gets RSSI value set in DRV_WIFI_RssiSet(). It does not retrieve the current connection RSSI value. The scan result will yield the current RSSI.

Remarks

None

Preconditions

Wi-Fi initialization must be complete

Example

```
uint8_t rssi;

DRV_WIFI_RssiGet(&rssi);
```

Parameters

Parameters	Description
p_rssi	pointer where rssi value is written

Function

void DRV_WIFI_RssiGet(uint8_t *p_rssi);

DRV_WIFI_isHibernateEnable Function

Checks if MRF24W is in hibernate mode.

Implementation: Dynamic

File

drv_wifi.h

C

```
bool DRV_WIFI_isHibernateEnable();
```

Returns

true or false

Description

Checks if MRF24W is in hibernate mode

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
bool flag;

flag = DRV_WIFI_isHibernateEnable();
```

Function

bool DRV_WIFI_isHibernateEnable(void)

DRV_WIFI_SecurityTypeGet Function

File

drv_wifi.h

C

```
void DRV_WIFI_SecurityTypeGet(uint8_t * p_securityType);
```

Description

This is function DRV_WIFI_SecurityTypeGet.

DRV_WIFI_TxPowerFactoryMaxGet Function

Retrieves the factory-set max TX power from the MRF24W.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_TxPowerFactoryMaxGet(uint8_t * p_factoryMaxTxPower);
```

Returns

None.

Description

This function retrieves the factory-set max TX power from the MRF24WG.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t maxPower;

DRV_WIFI_TxPowerFactoryMaxGet(&maxPower);
```

Parameters

Parameters	Description
p_factoryMaxTxPower	pointer to where factory max power is written (dbM)

Function

void DRV_WIFI_TxPowerFactoryMaxGet(int8_t *p_factoryMaxTxPower)

DRV_WIFI_TxPowerMaxGet Function

Gets the TX max power on the MRF24WG0M.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_TxPowerMaxGet(uint8_t * p_maxTxPower);
```

Returns

None.

Description

Gets the TX max power setting from the MRF24WG.

Remarks

None

Preconditions

Wi-Fi initialization must be complete.

Example

```
uint8_t maxPower;

DRV_WIFI_TxPowerMaxGet(&maxPower);
```

Parameters

Parameters	Description
p_maxTxPower	pointer to where max power setting is written (dBm).

Function

void DRV_WIFI_TxPowerMaxGet(uint8_t *p_maxTxPower);

DRV_WIFI_TxPowerMaxSet Function

Sets the TX max power on the MRF24WG0M.

Implementation: Dynamic

File

drv_wifi.h

C

```
void DRV_WIFI_TxPowerMaxSet(uint8_t maxTxPower);
```

Returns

None.

Description

After initialization the MRF24WG0M max TX power is determined by a factory-set value. This function can set a different maximum TX power levels. However, this function can never set a maximum TX power greater than the factory-set value, which can be read via DRV_WIFI_TxPowerFactoryMaxGet().

Remarks

No conversion of units needed, input to MRF24WG0M is in dBm.

Preconditions

Wi-Fi initialization must be complete.

Example

```
DRV_WIFI_TxPowerMaxSet(8); // set max TX power to 8dBm
```

Parameters

Parameters	Description
maxTxPower	valid range (0 to 17 dBm)

Function

void DRV_WIFI_TxPowerMaxSet(uint8_t maxTxPower)

m) Data Types and Constants

DRV_WIFI_BSSID_LENGTH Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_BSSID_LENGTH (6)
```

Description

This is macro DRV_WIFI_BSSID_LENGTH.

DRV_WIFI_DEAUTH_REASONCODE_MASK Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DEAUTH_REASONCODE_MASK ((uint8_t)0x80)
```

Description

This is macro DRV_WIFI_DEAUTH_REASONCODE_MASK.

DRV_WIFI_DEFAULT_ADHOC_BEACON_PERIOD Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_ADHOC_BEACON_PERIOD (100) // ms
```

Description

ms

DRV_WIFI_DEFAULT_ADHOC_HIDDEN_SSID Macro

Default values for Wi-Fi AdHoc settings

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_ADHOC_HIDDEN_SSID false
```

Description

Wi-Fi AdHoc default settings

These defines identify various default Wi-Fi AdHoc settings that can be used in the DRV_WIFI_ADHOC_NETWORK_CONTEXT structure.

DRV_WIFI_DEFAULT_ADHOC_MODE Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_ADHOC_MODE DRV_WIFI_ADHOC_CONNECT_THEN_START
```

Description

This is macro DRV_WIFI_DEFAULT_ADHOC_MODE.

DRV_WIFI_DEFAULT_PS_DTIM_ENABLED Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_PS_DTIM_ENABLED true                // DTIM wake-up enabled (normally
the case)
```

Description

DTIM wake-up enabled (normally the case)

DRV_WIFI_DEFAULT_PS_DTIM_INTERVAL Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_PS_DTIM_INTERVAL ((uint16_t)2)        // number of beacon periods
```

Description

number of beacon periods

DRV_WIFI_DEFAULT_PS_LISTEN_INTERVAL Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_PS_LISTEN_INTERVAL ((uint16_t)1)        // 100ms multiplier, e.g. 1 *
100ms = 100ms
```

Description

100ms multiplier, e.g. 1 * 100ms = 100ms

DRV_WIFI_DEFAULT_SCAN_COUNT Macro

Default values for Wi-Fi scan context

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_SCAN_COUNT (1)
```

Description

Wi-Fi Scan Context default settings

These defines identify the default Wi-Fi scan context values that can be used in the DRV_WIFI_SCAN_CONTEXT structure.

DRV_WIFI_DEFAULT_SCAN_MAX_CHANNEL_TIME Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_SCAN_MAX_CHANNEL_TIME (400)    // ms
```

Description

ms

DRV_WIFI_DEFAULT_SCAN_MIN_CHANNEL_TIME Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_SCAN_MIN_CHANNEL_TIME (200)   // ms
```

Description

ms

DRV_WIFI_DEFAULT_SCAN_PROBE_DELAY Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_SCAN_PROBE_DELAY (20)    // us
```

Description

us

DRV_WIFI_DEFAULT_WEP_KEY_TYPE Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_WEP_KEY_TYPE DRV_WIFI_SECURITY_WEP_OPENKEY
```

Description

This is macro DRV_WIFI_DEFAULT_WEP_KEY_TYPE.

DRV_WIFI_DISABLED Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DISABLED (0)
```

Section

Data Types and Constants

```
*****************************************************************************

*****************************************************************************
```

Do not make this an enumerated type!

DRV_WIFI_DISASSOC_REASONCODE_MASK Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DISASSOC_REASONCODE_MASK ((uint8_t)0x40)
```

Description

This is macro DRV_WIFI_DISASSOC_REASONCODE_MASK.

DRV_WIFI_ENABLED Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_ENABLED (1)
```

Description

This is macro DRV_WIFI_ENABLED.

DRV_WIFI_MAX_CHANNEL_LIST_LENGTH Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_MAX_CHANNEL_LIST_LENGTH (14)
```

Description

This is macro DRV_WIFI_MAX_CHANNEL_LIST_LENGTH.

DRV_WIFI_MAX_NUM_RATES Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_MAX_NUM_RATES (8)
```

Description

This is macro DRV_WIFI_MAX_NUM_RATES.

DRV_WIFI_MAX_SECURITY_KEY_LENGTH Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_MAX_SECURITY_KEY_LENGTH (64)
```

Description

This is macro DRV_WIFI_MAX_SECURITY_KEY_LENGTH.

© 2013-2015 Microchip Technology Inc.

DRV_WIFI_MAX_SSID_LENGTH Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_MAX_SSID_LENGTH (32)
```

Description

This is macro DRV_WIFI_MAX_SSID_LENGTH.

DRV_WIFI_MAX_WEP_KEY_LENGTH Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_MAX_WEP_KEY_LENGTH (DRV_WIFI_WEP104_KEY_LENGTH)
```

Description

This is macro DRV_WIFI_MAX_WEP_KEY_LENGTH.

DRV_WIFI_MAX_WPA_PASS_PHRASE_LENGTH Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_MAX_WPA_PASS_PHRASE_LENGTH (63) // must exclude string terminator
```

Description

must exclude string terminator

DRV_WIFI_MIN_WPA_PASS_PHRASE_LENGTH Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_MIN_WPA_PASS_PHRASE_LENGTH (8) // must exclude string terminator
```

Description

must exclude string terminator

DRV_WIFI_NETWORK_TYPE_ADHOC Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_NETWORK_TYPE_ADHOC (2)
```

Description

This is macro DRV_WIFI_NETWORK_TYPE_ADHOC.

DRV_WIFI_NETWORK_TYPE_INFRASTRUCTURE Macro

Selection of different Wi-Fi network types

File

drv_wifi.h

C

```
#define DRV_WIFI_NETWORK_TYPE_INFRASTRUCTURE (1)
```

Description

Wi-Fi Network Types

This enumeration identifies the Wi-Fi network types that can be selected. Do NOT make these an enumerated type as they are used as a compile switch.

DRV_WIFI_NETWORK_TYPE_P2P Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_NETWORK_TYPE_P2P (3) /* not supported */
```

Description

not supported

DRV_WIFI_NETWORK_TYPE_SOFT_AP Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_NETWORK_TYPE_SOFT_AP (4)
```

Description

This is macro DRV_WIFI_NETWORK_TYPE_SOFT_AP.

DRV_WIFI_NO_ADDITIONAL_INFO Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_NO_ADDITIONAL_INFO ((uint16_t)0xffff)
```

Description

eventInfo define for DRV_WIFI_ProcessEvent() when no additional info is supplied

DRV_WIFI_RETRY_ADHOC Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_RETRY_ADHOC (3)
```

Description

This is macro DRV_WIFI_RETRY_ADHOC.

DRV_WIFI_RETRY_FOREVER Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_RETRY_FOREVER (255)
```

Description

This is macro DRV_WIFI_RETRY_FOREVER.

DRV_WIFI_RTS_THRESHOLD_MAX Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_RTS_THRESHOLD_MAX (2347) /* maximum RTS threshold size in bytes */
```

Description

maximum RTS threshold size in bytes

DRV_WIFI_SECURITY_EAP Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_SECURITY_EAP (11) /* not supported */
```

Description

not supported

DRV_WIFI_SECURITY_OPEN Macro

Selection of different Wi-Fi security types

File

drv_wifi.h

C

```
#define DRV_WIFI_SECURITY_OPEN (0)
```

Description

Wi-Fi Security Types

This enumeration identifies the Wi-Fi security types that can be selected. Do NOT make these an enumerated type as they are used as a compile switch.

DRV_WIFI_SECURITY_WEP_104 Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_SECURITY_WEP_104 (2)
```

Description

This is macro DRV_WIFI_SECURITY_WEP_104.

DRV_WIFI_SECURITY_WEP_40 Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_SECURITY_WEP_40 (1)
```

Description

This is macro DRV_WIFI_SECURITY_WEP_40.

DRV_WIFI_SECURITY_WPA_AUTO_WITH_KEY Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_SECURITY_WPA_AUTO_WITH_KEY (7)
```

Description

This is macro DRV_WIFI_SECURITY_WPA_AUTO_WITH_KEY.

DRV_WIFI_SECURITY_WPA_AUTO_WITH_PASS_PHRASE Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_SECURITY_WPA_AUTO_WITH_PASS_PHRASE (8)
```

Description

This is macro DRV_WIFI_SECURITY_WPA_AUTO_WITH_PASS_PHRASE.

© 2013-2015 Microchip Technology Inc.

DRV_WIFI_SECURITY_WPS_PIN Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_SECURITY_WPS_PIN (10)
```

Description

This is macro DRV_WIFI_SECURITY_WPS_PIN.

DRV_WIFI_SECURITY_WPS_PUSH_BUTTON Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_SECURITY_WPS_PUSH_BUTTON (9)
```

Description

This is macro DRV_WIFI_SECURITY_WPS_PUSH_BUTTON.

DRV_WIFI_WEP104_KEY_LENGTH Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_WEP104_KEY_LENGTH (52) // 4 keys of 13 bytes each
```

Description

4 keys of 13 bytes each

DRV_WIFI_WEP40_KEY_LENGTH Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_WEP40_KEY_LENGTH (20) // 4 keys of 5 bytes each
```

Description

4 keys of 5 bytes each

DRV_WIFI_WPA_KEY_LENGTH Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_WPA_KEY_LENGTH (32)
```

Description

This is macro DRV_WIFI_WPA_KEY_LENGTH.

© 2013-2015 Microchip Technology Inc.

DRV_WIFI_WPS_PIN_LENGTH Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_WPS_PIN_LENGTH (8) // 7 digits + checksum byte
```

Description

7 digits + checksum byte

DRV_WIFI_ADHOC_MODES Enumeration

Selection of different AdHoc connection modes

File

drv_wifi.h

C

```
typedef enum adhocMode {
  DRV_WIFI_ADHOC_CONNECT_THEN_START = 0,
  DRV_WIFI_ADHOC_CONNECT_ONLY = 1,
  DRV_WIFI_ADHOC_START_ONLY = 2
} DRV_WIFI_ADHOC_MODES;
```

Members

Members	Description
DRV_WIFI_ADHOC_CONNECT_THEN_START = 0	try to connect existing AdHoc network, if not found then start network
DRV_WIFI_ADHOC_CONNECT_ONLY = 1	only connect to existing AdHoc network
DRV_WIFI_ADHOC_START_ONLY = 2	only start a new AdHoc network

Description

AdHoc Modes

This enumeration identifies the AdHoc modes that can be selected when connecting in AdHoc mode.

DRV_WIFI_ADHOC_NETWORK_CONTEXT Structure

Contains data pertaining to Wi-Fi AdHoc context

File

drv_wifi.h

C

```
typedef struct {
  uint8_t mode;
  bool hiddenSsid;
  uint16_t beaconPeriod;
} DRV_WIFI_ADHOC_NETWORK_CONTEXT;
```

Members

Members	Description
uint8_t mode;	Defines how to start the AdHoc network. See DRV_WIFI_ADHOC_MODE. Default is DRV_WIFI_ADHOC_CONNECT_THEN_START.
bool hiddenSsid;	When starting an AdHoc network, the SSID can be hidden in the beacons. Set true to hide the SSID, else false. Default is false.
uint16_t beaconPeriod;	Sets the beacon period, in ms. Default is 100ms

Description

Wi-Fi AdHoc Context

This structure contains MRF24WG AdHoc context data. See DRV_WIFI_AdhocContextSet().

DRV_WIFI_CONNECTION_CONTEXT Structure

Contains data pertaining to MRF24WG connection context

File

drv_wifi.h

C

```
typedef struct {
  uint8_t channel;
  uint8_t bssid[6];
} DRV_WIFI_CONNECTION_CONTEXT;
```

Members

Members	Description
uint8_t channel;	channel number of current connection
uint8_t bssid[6];	bssid of connected AP

Description

Wi-Fi Connection Context

This structure contains MRF24WG connection context data. See DRV_WIFI_ConnectContextGet().

DRV_WIFI_CONNECTION_STATES Enumeration

Wi-Fi Connection states

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_CSTATE_NOT_CONNECTED = 1,
  DRV_WIFI_CSTATE_CONNECTION_IN_PROGRESS = 2,
  DRV_WIFI_CSTATE_CONNECTED_INFRASTRUCTURE = 3,
  DRV_WIFI_CSTATE_CONNECTED_ADHOC = 4,
  DRV_WIFI_CSTATE_RECONNECTION_IN_PROGRESS = 5,
  DRV_WIFI_CSTATE_CONNECTION_PERMANENTLY_LOST = 6
} DRV_WIFI_CONNECTION_STATES;
```

Members

Members	Description
DRV_WIFI_CSTATE_NOT_CONNECTED = 1	No Wi-Fi connection exists
DRV_WIFI_CSTATE_CONNECTION_IN_PROGRESS = 2	Wi-Fi connection in progress
DRV_WIFI_CSTATE_CONNECTED_INFRASTRUCTURE = 3	Wi-Fi connected in infrastructure mode
DRV_WIFI_CSTATE_CONNECTED_ADHOC = 4	Wi-Fi connected in adHoc mode
DRV_WIFI_CSTATE_RECONNECTION_IN_PROGRESS = 5	Wi-Fi in process of reconnecting
DRV_WIFI_CSTATE_CONNECTION_PERMANENTLY_LOST = 6	Wi-Fi connection permanently lost

Description

Wi-Fi Connection States

This enumeration identifies Wi-Fi Connection states. See DRV_WIFI_ConnectionStateGet().

DRV_WIFI_DEVICE_INFO Structure

Contains data pertaining to MRF24WG device type and version number

File

drv_wifi.h

C

```
typedef struct {
  uint8_t deviceType;
  uint8_t romVersion;
  uint8_t patchVersion;
} DRV_WIFI_DEVICE_INFO;
```

Members

Members	Description
uint8_t deviceType;	MRF24W_DEVICE_TYPE
uint8_t romVersion;	MRF24WG ROM version number
uint8_t patchVersion;	MRF24WG patch version number

Description

Wi-Fi Device Type/Version

This structure contains MRF24WG device type and version number. See DRV_WIFI_DeviceInfoGet().

DRV_WIFI_DEVICE_TYPES Enumeration

Codes for Wi-Fi device types

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_MRF24WB0M_DEVICE = 1,
  DRV_WIFI_MRF24WG0M_DEVICE = 2
} DRV_WIFI_DEVICE_TYPES;
```

Description

Wi-Fi devices types

This enumeration identifies Wi-Fi device types. The only device supported with this driver is DRV_WIFI_MRF24WG0M_DEVICE

DRV_WIFI_DOMAIN_CODES Enumeration

Regional domain codes.

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_DOMAIN_FCC = 0,
  DRV_WIFI_DOMAIN_ETSI = 2,
  DRV_WIFI_DOMAIN_JAPAN = 7,
  DRV_WIFI_DOMAIN_OTHER = 7
} DRV_WIFI_DOMAIN_CODES;
```

Members

Members	Description
DRV_WIFI_DOMAIN_FCC = 0	FCC, available channels: 1 - 11
DRV_WIFI_DOMAIN_ETSI = 2	ESTI, available Channels: 1 - 13
DRV_WIFI_DOMAIN_JAPAN = 7	Japan, available Channels: 1 - 14
DRV_WIFI_DOMAIN_OTHER = 7	Other, available Channels: 1 - 14

Description

Wi-Fi regional domain codes

This enumeration identifies Wi-Fi regional domain codes. The regional domain can be determined by calling DRV_WIFI_RegionalDomainGet().

DRV_WIFI_EVENT_CONN_TEMP_LOST_CODES Enumeration

Selection of different codes when Wi-Fi connection is temporarily lost.

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_BEACON_TIMEOUT = 1,
  DRV_WIFI_DEAUTH_RECEIVED = 2,
  DRV_WIFI_DISASSOCIATE_RECEIVED = 3
} DRV_WIFI_EVENT_CONN_TEMP_LOST_CODES;
```

Members

Members	Description
DRV_WIFI_BEACON_TIMEOUT = 1	connection temporarily lost due to beacon timeout
DRV_WIFI_DEAUTH_RECEIVED = 2	connection temporarily lost due to deauthorization received from AP
DRV_WIFI_DISASSOCIATE_RECEIVED = 3	connection temporarily lost due to disassociation received from AP

Description

'Connection Temporarily Lost' event codes

This enumeration identifies the codes for a connection temporarily lost. These codes are used in DRV_WIFI_ProcessEvent(), case DRV_WIFI_EVENT_CONNECTION_TEMPORARILY_LOST.

DRV_WIFI_EVENT_INFO Enumeration

Selection of different EventInfo types

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_JOIN_FAILURE = 2,
  DRV_WIFI_AUTHENTICATION_FAILURE = 3,
  DRV_WIFI_ASSOCIATION_FAILURE = 4,
  DRV_WIFI_WEP_HANDSHAKE_FAILURE = 5,
  DRV_WIFI_PSK_CALCULATION_FAILURE = 6,
  DRV_WIFI_PSK_HANDSHAKE_FAILURE = 7,
  DRV_WIFI_ADHOC_JOIN_FAILURE = 8,
  DRV_WIFI_SECURITY_MISMATCH_FAILURE = 9,
  DRV_WIFI_NO_SUITABLE_AP_FOUND_FAILURE = 10,
  DRV_WIFI_RETRY_FOREVER_NOT_SUPPORTED_FAILURE = 11,
  DRV_WIFI_LINK_LOST = 12,
  DRV_WIFI_TKIP_MIC_FAILURE = 13,
  DRV_WIFI_RSN_MIXED_MODE_NOT_SUPPORTED = 14,
  DRV_WIFI_RECV_DEAUTH = 15,
  DRV_WIFI_RECV_DISASSOC = 16,
  DRV_WIFI_WPS_FAILURE = 17,
  DRV_WIFI_P2P_FAILURE = 18,
  DRV_WIFI_LINK_DOWN = 19
} DRV_WIFI_EVENT_INFO;
```

Members

Members	Description
DRV_WIFI_P2P_FAILURE = 18	not supported

Description

EventInfo types

This enumeration identifies the eventInfo types used in DRV_WIFI_ProcessEvent(), case DRV_WIFI_EVENT_CONNECTION_FAILED.

DRV_WIFI_EVENTS Enumeration

Selections for events that can occur.

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_EVENT_NONE = 0,
  DRV_WIFI_EVENT_CONNECTION_SUCCESSFUL = 1,
  DRV_WIFI_EVENT_CONNECTION_FAILED = 2,
  DRV_WIFI_EVENT_CONNECTION_TEMPORARILY_LOST = 3,
  DRV_WIFI_EVENT_CONNECTION_PERMANENTLY_LOST = 4,
  DRV_WIFI_EVENT_CONNECTION_REESTABLISHED = 5,
  DRV_WIFI_EVENT_FLASH_UPDATE_SUCCESSFUL = 6,
  DRV_WIFI_EVENT_FLASH_UPDATE_FAILED = 7,
  DRV_WIFI_EVENT_KEY_CALCULATION_REQUEST = 8,
  DRV_WIFI_EVENT_INVALID_WPS_PIN = 9,
  DRV_WIFI_EVENT_SCAN_RESULTS_READY = 10,
  DRV_WIFI_EVENT_IE_RESULTS_READY = 11,
  DRV_WIFI_EVENT_SOFT_AP = 12,
  DRV_WIFI_EVENT_DISCONNECT_DONE = 13,
  DRV_WIFI_EVENT_UPDATE = 14,
  DRV_WIFI_EVENT_ERROR = 15
} DRV_WIFI_EVENTS;
```

Members

Members	Description
DRV_WIFI_EVENT_NONE = 0	No event has occurred
DRV_WIFI_EVENT_CONNECTION_SUCCESSFUL = 1	Connection attempt to network successful
DRV_WIFI_EVENT_CONNECTION_FAILED = 2	Connection attempt failed
DRV_WIFI_EVENT_CONNECTION_TEMPORARILY_LOST = 3	Connection lost; MRF24W attempting to reconnect
DRV_WIFI_EVENT_CONNECTION_PERMANENTLY_LOST = 4	Connection lost; MRF24W no longer trying to connect
DRV_WIFI_EVENT_CONNECTION_REESTABLISHED = 5	Connection has been reestablished
DRV_WIFI_EVENT_FLASH_UPDATE_SUCCESSFUL = 6	Update to FLASH successful
DRV_WIFI_EVENT_FLASH_UPDATE_FAILED = 7	Update to FLASH failed
DRV_WIFI_EVENT_KEY_CALCULATION_REQUEST = 8	Key calculation is required
DRV_WIFI_EVENT_INVALID_WPS_PIN = 9	Invalid WPS pin was entered
DRV_WIFI_EVENT_SCAN_RESULTS_READY = 10	Scan results are ready
DRV_WIFI_EVENT_IE_RESULTS_READY = 11	IE data ready
DRV_WIFI_EVENT_SOFT_AP = 12	Client connection events
DRV_WIFI_EVENT_DISCONNECT_DONE = 13	Disconnect done event
DRV_WIFI_EVENT_UPDATE = 14	Wi-Fi update event occurred
DRV_WIFI_EVENT_ERROR = 15	Wi-Fi error event occurred

Description

Wi-Fi Events

This enumeration identifies the Wi-Fi events that can occur and will be sent to DRV_WIFI_ProcessEvent().

DRV_WIFI_GENERAL_ERRORS Enumeration

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_ERROR_IN_HIBERNATE_MODE = 100
} DRV_WIFI_GENERAL_ERRORS;
```

Members

Members	Description
DRV_WIFI_ERROR_IN_HIBERNATE_MODE = 100	invalid operation while MRF24WG is in hibernate mode

Description

This is type DRV_WIFI_GENERAL_ERRORS.

DRV_WIFI_HIBERNATE_STATES Enumeration

Wi-Fi Hibernate states

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_HB_NO_SLEEP = 0,
  DRV_WIFI_HB_ENTER_SLEEP = 1,
  DRV_WIFI_HB_WAIT_WAKEUP = 2
} DRV_WIFI_HIBERNATE_STATES;
```

Description

Wi-Fi Hibernate states

This enumeration identifies Wi-Fi hibernate states.

DRV_WIFI_MAC_STATS Structure

Wi-Fi MIB states

File

drv_wifi.h

C

```
typedef struct {
  uint32_t MibWEPExcludeCtr;
  uint32_t MibTxBytesCtr;
  uint32_t MibTxMulticastCtr;
  uint32_t MibTxFailedCtr;
  uint32_t MibTxRtryCtr;
  uint32_t MibTxMultRtryCtr;
  uint32_t MibTxSuccessCtr;
  uint32_t MibRxDupCtr;
  uint32_t MibRxCtsSuccCtr;
  uint32_t MibRxCtsFailCtr;
  uint32_t MibRxAckFailCtr;
  uint32_t MibRxBytesCtr;
  uint32_t MibRxFragCtr;
  uint32_t MibRxMultCtr;
  uint32_t MibRxFCSErrCtr;
  uint32_t MibRxWEPUndecryptCtr;
  uint32_t MibRxFragAgedCtr;
  uint32_t MibRxMICFailureCtr;
} DRV_WIFI_MAC_STATS;
```

Members

Members	Description
uint32_t MibWEPExcludeCtr;	Number of frames received with the Protected Frame sub-field of the Frame Control field set to zero and the value of dot11ExcludeUnencrypted causes that frame to be discarded.
uint32_t MibTxBytesCtr;	Total number of TX bytes that have been transmitted
uint32_t MibTxMulticastCtr;	Number of frames successfully transmitted that had the multicast bit set in the destination MAC address
uint32_t MibTxFailedCtr;	Number of TX frames that failed due to the number of transmits exceeding the retry count
uint32_t MibTxRtryCtr;	Number of times a transmitted frame needed to be retried.
uint32_t MibTxMultRtryCtr;	Number of times a frame was successfully transmitted after more than one retransmission.
uint32_t MibTxSuccessCtr;	Number of TX frames successfully transmitted.
uint32_t MibRxDupCtr;	Number of frames received where the Sequence Control field indicates a duplicate.
uint32_t MibRxCtsSuccCtr;	Number of CTS frames received in response to an RTS frame.
uint32_t MibRxCtsFailCtr;	Number of times an RTS frame was not received in response to a CTS frame.
uint32_t MibRxAckFailCtr;	Number of times an ACK was not received in response to a TX frame.
uint32_t MibRxBytesCtr;	Total number of Rx bytes received.
uint32_t MibRxFragCtr;	Number of successful received frames (management or data).
uint32_t MibRxMultCtr;	Number of frames received with the multicast bit set in the destination MAC address.
uint32_t MibRxFCSErrCtr;	Number of frames received with an invalid Frame Checksum (FCS).

uint32_t MibRxWEPUndecryptCtr;	Number of frames received where the Protected Frame sub-field of the Frame Control Field is set to one and the WEPOn value for the key mapped to the transmitter's MAC address indicates the frame should not have been encrypted.
uint32_t MibRxFragAgedCtr;	Number of times that fragments aged out, or were not received in the allowable time.
uint32_t MibRxMICFailureCtr;	Number of MIC failures that have occurred.

Description

Wi-Fi MIB states

This structure contains all the MIB data returned from the MRF24WG when DRV_WIFI_MacStatsGet() is called.

DRV_WIFI_MGMT_ERRORS Enumeration

Error codes returned when a management message is sent to the MRF24W

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_SUCCESS = 1,
  DRV_WIFI_ERROR_INVALID_SUBTYPE = 2,
  DRV_WIFI_ERROR_OPERATION_CANCELLED = 3,
  DRV_WIFI_ERROR_FRAME_END_OF_LINE_OCCURRED = 4,
  DRV_WIFI_ERROR_FRAME_RETRY_LIMIT_EXCEEDED = 5,
  DRV_WIFI_ERROR_EXPECTED_BSS_VALUE_NOT_IN_FRAME = 6,
  DRV_WIFI_ERROR_FRAME_SIZE_EXCEEDS_BUFFER_SIZE = 7,
  DRV_WIFI_ERROR_FRAME_ENCRYPT_FAILED = 8,
  DRV_WIFI_ERROR_INVALID_PARAM = 9,
  DRV_WIFI_ERROR_AUTH_REQ_ISSUED_WHILE_IN_AUTH_STATE = 10,
  DRV_WIFI_ERROR_ASSOC_REQ_ISSUED_WHILE_IN_ASSOC_STATE = 11,
  DRV_WIFI_ERROR_INSUFFICIENT_RESOURCES = 12,
  DRV_WIFI_ERROR_TIMEOUT_OCCURRED = 13,
  DRV_WIFI_ERROR_BAD_EXCHANGE_ENCOUNTERED_IN_FRAME_RECEPTION = 14,
  DRV_WIFI_ERROR_AUTH_REQUEST_REFUSED = 15,
  DRV_WIFI_ERROR_ASSOCIATION_REQUEST_REFUSED = 16,
  DRV_WIFI_ERROR_PRIOR_MGMT_REQUEST_IN_PROGRESS = 17,
  DRV_WIFI_ERROR_NOT_IN_JOINED_STATE = 18,
  DRV_WIFI_ERROR_NOT_IN_ASSOCIATED_STATE = 19,
  DRV_WIFI_ERROR_NOT_IN_AUTHENTICATED_STATE = 20,
  DRV_WIFI_ERROR_SUPPLICANT_FAILED = 21,
  DRV_WIFI_ERROR_UNSUPPORTED_FEATURE = 22,
  DRV_WIFI_ERROR_REQUEST_OUT_OF_SYNC = 23,
  DRV_WIFI_ERROR_CP_INVALID_ELEMENT_TYPE = 24,
  DRV_WIFI_ERROR_CP_INVALID_PROFILE_ID = 25,
  DRV_WIFI_ERROR_CP_INVALID_DATA_LENGTH = 26,
  DRV_WIFI_ERROR_CP_INVALID_SSID_LENGTH = 27,
  DRV_WIFI_ERROR_CP_INVALID_SECURITY_TYPE = 28,
  DRV_WIFI_ERROR_CP_INVALID_SECURITY_KEY_LENGTH = 29,
  DRV_WIFI_ERROR_CP_INVALID_WEP_KEY_ID = 30,
  DRV_WIFI_ERROR_CP_INVALID_NETWORK_TYPE = 31,
  DRV_WIFI_ERROR_CP_INVALID_ADHOC_MODE = 32,
  DRV_WIFI_ERROR_CP_INVALID_SCAN_TYPE = 33,
  DRV_WIFI_ERROR_CP_INVALID_CP_LIST = 34,
  DRV_WIFI_ERROR_CP_INVALID_CHANNEL_LIST_LENGTH = 35,
  DRV_WIFI_ERROR_NOT_CONNECTED = 36,
  DRV_WIFI_ERROR_ALREADY_CONNECTING = 37,
  DRV_WIFI_ERROR_DISCONNECT_FAILED = 38,
  DRV_WIFI_ERROR_NO_STORED_BSS_DESCRIPTOR = 39,
  DRV_WIFI_ERROR_INVALID_MAX_POWER = 40,
  DRV_WIFI_ERROR_CONNECTION_TERMINATED = 41,
  DRV_WIFI_ERROR_HOST_SCAN_NOT_ALLOWED = 42,
  DRV_WIFI_ERROR_INVALID_WPS_PIN = 44
} DRV_WIFI_MGMT_ERRORS;
```

Members

Members	Description
DRV_WIFI_ERROR_DISCONNECT_FAILED = 38	Disconnect failed. Disconnect is allowed only when module is in connected state
DRV_WIFI_ERROR_NO_STORED_BSS_DESCRIPTOR = 39	No stored scan results
DRV_WIFI_ERROR_HOST_SCAN_NOT_ALLOWED = 42	Host Scan Failed. Host scan is allowed only in idle or connected state

DRV_WIFI_ERROR_INVALID_WPS_PIN = 44	WPS pin was invalid

Description

Management Message Error Codes

This enumeration identifies the errors that can occur when a DRV_WIFI API function call results in a management message being sent, via SPI, to the MRF24W.

DRV_WIFI_MGMT_INDICATE_SOFT_AP_EVENT Structure

Contains data pertaining to Wi-Fi Soft AP event

File

drv_wifi.h

C

```
typedef struct {
  uint8_t reason;
  uint8_t address[6];
} DRV_WIFI_MGMT_INDICATE_SOFT_AP_EVENT;
```

Members

Members	Description
uint8_t reason;	reason code
uint8_t address[6];	MAC address

Description

Wi-Fi Soft AP Event Information

This structure contains data pertaining to Soft AP event. See DRV_WIFI_SoftApEventInfoGet().

DRV_WIFI_MULTICAST_FILTER_IDS Enumeration

Selections for software Multicast filter IDs

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_MULTICAST_FILTER_1 = 4,
  DRV_WIFI_MULTICAST_FILTER_2 = 5,
  DRV_WIFI_MULTICAST_FILTER_3 = 6,
  DRV_WIFI_MULTICAST_FILTER_4 = 7,
  DRV_WIFI_MULTICAST_FILTER_5 = 8,
  DRV_WIFI_MULTICAST_FILTER_6 = 9,
  DRV_WIFI_MULTICAST_FILTER_7 = 10,
  DRV_WIFI_MULTICAST_FILTER_8 = 11,
  DRV_WIFI_MULTICAST_FILTER_9 = 12,
  DRV_WIFI_MULTICAST_FILTER_10 = 13,
  DRV_WIFI_MULTICAST_FILTER_11 = 14,
  DRV_WIFI_MULTICAST_FILTER_12 = 15,
  DRV_WIFI_MULTICAST_FILTER_13 = 16,
  DRV_WIFI_MULTICAST_FILTER_14 = 17,
  DRV_WIFI_MULTICAST_FILTER_15 = 18,
  DRV_WIFI_MULTICAST_FILTER_16 = 19
} DRV_WIFI_MULTICAST_FILTER_IDS;
```

Description

Multicast Filter IDs

This enumeration identifies the multicast filters that can be selected. See DRV_WIFI_SWMulticastFilterSet().

DRV_WIFI_MULTICAST_FILTERS Enumeration

Selections for Software Multicast Filters.

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_MULTICAST_DISABLE_ALL = 0,
  DRV_WIFI_MULTICAST_ENABLE_ALL = 1,
  DRV_WIFI_MULTICAST_USE_FILTERS = 2
} DRV_WIFI_MULTICAST_FILTERS;
```

Members

Members	Description
DRV_WIFI_MULTICAST_DISABLE_ALL = 0	Discard all received multicast messages
DRV_WIFI_MULTICAST_ENABLE_ALL = 1	forward all multicast messages to host MCU
DRV_WIFI_MULTICAST_USE_FILTERS = 2	Use the MAC filtering capability for multicast messages

Description

Multicast Filter Modes

This enumeration identifies the mode of multicast filters that can be selected. See DRV_WIFI_SWMulticastFilterSet().

DRV_WIFI_POWER_SAVE_STATES Enumeration

Wi-Fi Power-Saving states

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_PS_HIBERNATE = 1,
  DRV_WIFI_PS_PS_POLL_DTIM_ENABLED = 2,
  DRV_WIFI_PS_PS_POLL_DTIM_DISABLED = 3,
  DRV_WIFI_PS_OFF = 4
} DRV_WIFI_POWER_SAVE_STATES;
```

Members

Members	Description
DRV_WIFI_PS_HIBERNATE = 1	enable hibernate mode
DRV_WIFI_PS_PS_POLL_DTIM_ENABLED = 2	enable power-saving mode with DTIM enabled
DRV_WIFI_PS_PS_POLL_DTIM_DISABLED = 3	enable power-saving mode with DTIM disabled
DRV_WIFI_PS_OFF = 4	disable power-saving mode

Description

Wi-Fi Power-Saving states

This enumeration identifies Wi-Fi Power-Saving states. See DRV_WIFI_PsPollEnable().

DRV_WIFI_PS_POLL_CONTEXT Structure

Contains data pertaining to Wi-Fi PS-Poll context

File

drv_wifi.h

C

```
typedef struct {
  uint16_t listenInterval;
  uint16_t dtimInterval;
  bool useDtim;
} DRV_WIFI_PS_POLL_CONTEXT;
```

Members

Members	Description
uint16_t listenInterval;	Number of 100ms intervals between instances when the MRF24WG wakes up to received buffered messages from the network. Each count represents 100ms. For example, 1 = 100ms, 2 = 200ms, etc. The default is 1 (100ms).
uint16_t dtimInterval;	Only used if useDtim is true. The DTIM period indicates how often clients serviced by the access point should check for buffered multicast or broadcast messages awaiting pickup on the access point. The DTIM interval is measured in number of beacon periods. Default for DTIM period is 2.
bool useDtim;	True: (default) check for buffered multicast or broadcast messages on the dtimInterval. False: check for buffered multicast or broadcast messages on the listenInterval

Description

Wi-Fi PS-Poll Context

This structure contains MRF24WG PS-Poll context data. See DRV_WIFI_PsPollEnable().

DRV_WIFI_REASON_CODES Enumeration

Selection of different codes when a deauthorization or disassociation event has occurred.

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_UNSPECIFIED = 1,
  DRV_WIFI_REASON_PREV_AUTH_NOT_VALID = 2,
  DRV_WIFI_DEAUTH_LEAVING = 3,
  DRV_WIFI_DISASSOC_DUE_TO_INACTIVITY = 4,
  DRV_WIFI_DISASSOC_AP_BUSY = 5,
  DRV_WIFI_CLASS2_FRAME_FROM_NONAUTH_STA = 6,
  DRV_WIFI_CLASS3_FRAME_FROM_NONASSOC_STA = 7,
  DRV_WIFI_DISASSOC_STA_HAS_LEFT = 8,
  DRV_WIFI_STA_REQ_ASSOC_WITHOUT_AUTH = 9,
  DRV_WIFI_INVALID_IE = 13,
  DRV_WIFI_MIC_FAILURE = 14,
  DRV_WIFI_4WAY_HANDSHAKE_TIMEOUT = 15,
  DRV_WIFI_GROUP_KEY_HANDSHAKE_TIMEOUT = 16,
  DRV_WIFI_IE_DIFFERENT = 17,
  DRV_WIFI_INVALID_GROUP_CIPHER = 18,
  DRV_WIFI_INVALID_PAIRWISE_CIPHER = 19,
  DRV_WIFI_INVALID_AKMP = 20,
  DRV_WIFI_UNSUPP_RSN_VERSION = 21,
  DRV_WIFI_INVALID_RSN_IE_CAP = 22,
  DRV_WIFI_IEEE8021X_FAILED = 23,
  DRV_WIFI_CIPHER_SUITE_REJECTED = 24
} DRV_WIFI_REASON_CODES;
```

Description

Deauthorization/Disassociate Reason Codes

This enumeration identifies the reason codes for a connection lost due to a deauthorization or disassociation from the AP.

DRV_WIFI_RECONNECT_MODES Enumeration

Selection of different Reconnection modes

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_DO_NOT_ATTEMPT_TO_RECONNECT = 0,
  DRV_WIFI_ATTEMPT_TO_RECONNECT = 1
} DRV_WIFI_RECONNECT_MODES;
```

Description

Wi-Fi Reconnect Modes

This enumeration identifies the reconnection modes that can be used in DRV_WIFI_ReconnectModeSet().

DRV_WIFI_SCAN_CONTEXT Structure

Contains data pertaining to Wi-Fi scan context

File

drv_wifi.h

C

```
typedef struct {
  uint8_t scanType;
  uint8_t scanCount;
  uint16_t minChannelTime;
  uint16_t maxChannelTime;
  uint16_t probeDelay;
} DRV_WIFI_SCAN_CONTEXT;
```

Members

Members	Description
uint8_t scanType;	802.11 allows for active scanning, where the device sends out a broadcast probe request seeking an access point. Also allowed is passive scanning where the device only listens to beacons being broadcast from access points. Set to DRV_WIFI_ACTIVE_SCAN (default) or DRV_WIFI_PASSIVE_SCAN
uint8_t scanCount;	The number of times to scan a channel while attempting to find a particular access point. Default is 1
uint16_t minChannelTime;	The minimum time (in milliseconds) the MRF24WG will wait for a probe response after sending a probe request. If no probe responses are received in minChannelTime, the MRF24WG will go on to the next channel, if any are left to scan, or quit. Default is 200ms.
uint16_t maxChannelTime;	If a probe response is received within minChannelTime, the MRF24WG will continue to collect any additional probe responses up to maxChannelTime before going to the next channel in the channelList. Units are in milliseconds. Default is 400ms.
uint16_t probeDelay;	The number of microseconds to delay before transmitting a probe request following the channel change during scanning. Default is 20uS.

Description

Wi-Fi Scan Context

This structure contains MRF24WG scan context data. See DRV_WIFI_ScanContextSet().

DRV_WIFI_SCAN_RESULT Structure

Contains data pertaining to Wi-Fi scan results

File

drv_wifi.h

C

```
typedef struct {
  uint8_t bssid[DRV_WIFI_BSSID_LENGTH];
  uint8_t ssid[DRV_WIFI_MAX_SSID_LENGTH];
  uint8_t apConfig;
  uint8_t reserved;
  uint16_t beaconPeriod;
  uint16_t atimWindow;
  uint8_t basicRateSet[DRV_WIFI_MAX_NUM_RATES];
  uint8_t rssi;
  uint8_t numRates;
  uint8_t DtimPeriod;
  uint8_t bssType;
  uint8_t channel;
  uint8_t ssidLen;
} DRV_WIFI_SCAN_RESULT;
```

Members

Members	Description
uint8_t bssid[DRV_WIFI_BSSID_LENGTH];	Network BSSID value
uint8_t ssid[DRV_WIFI_MAX_SSID_LENGTH];	Network SSID value
uint8_t apConfig;	Access Point config (see description)
uint8_t reserved;	not used
uint16_t beaconPeriod;	Network beacon interval
uint16_t atimWindow;	Only valid if bssType = DRV_WIFI_NETWORK_TYPE_INFRASTRUCTURE
uint8_t basicRateSet[DRV_WIFI_MAX_NUM_RATES];	List of Network basic rates. Each rate has the following format: Bit 7 • 0: rate is not part of the basic rates set • 1: rate is part of the basic rates set Bits 6:0 Multiple of 500kbps giving the supported rate. For example, a value of 2 (2 * 500kbps) indicates that 1mbps is a supported rate. A value of 4 in this field indicates a 2mbps rate (4 * 500kbps).
uint8_t rssi;	Signal strength of received frame beacon or probe response. Will range from a low of 43 to a high of 128.
uint8_t numRates;	Number of valid rates in basicRates
uint8_t DtimPeriod;	Part of TIM element
uint8_t bssType;	DRV_WIFI_NETWORK_TYPE_INFRASTRUCTURE or DRV_WIFI_NETWORK_TYPE_ADHOC
uint8_t channel;	Channel number
uint8_t ssidLen;	Number of valid characters in ssid

Description

Wi-Fi Scan Results

This structure contains the result of Wi-Fi scan operation. See DRV_WIFI_ScanGetResult().

apConfig Bit Mask

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
WPA2	WPA	Preamble	Privacy	Reserved	Reserved	Reserved	IE

IE	1 if AP broadcasting one or more Information Elements, else 0
Privacy	0 : AP is open (no security) 1: AP using security, if neither WPA and WPA2 set then security is WEP.
Preamble	0: AP transmitting with short preamble 1: AP transmitting with long preamble
WPA	Only valid if Privacy is 1. 0: AP does not support WPA 1: AP supports WPA
WPA2	Only valid if Privacy is 1. 0: AP does not support WPA2 1: AP supports WPA2

DRV_WIFI_SCAN_TYPES Enumeration

Selection of different Wi-Fi scan types

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_ACTIVE_SCAN = 1,
  DRV_WIFI_PASSIVE_SCAN = 2
} DRV_WIFI_SCAN_TYPES;
```

Description

Wi-Fi Scan Types

This enumeration identifies the Wi-Fi scan types that can be selected.

DRV_WIFI_SECURITY_CONTEXT Union

Contains data pertaining to Wi-Fi security.

File

drv_wifi.h

C

```
typedef union {
  DRV_WIFI_WEP_CONTEXT wepContext;
  DRV_WIFI_WPA_CONTEXT wpaContext;
  DRV_WIFI_WPS_CONTEXT wpsContext;
} DRV_WIFI_SECURITY_CONTEXT;
```

Members

Members	Description
DRV_WIFI_WEP_CONTEXT wepContext;	set WEP security context
DRV_WIFI_WPA_CONTEXT wpaContext;	set WPA security context
DRV_WIFI_WPS_CONTEXT wpsContext;	set WPS security context

Description

Wi-Fi security context

Structure/union can be used in functions DRV_WIFI_SecurityWepSet, DRV_WIFI_SecurityWpaSet, and DRV_WIFI_SecurityWpsSet

DRV_WIFI_SOFT_AP_EVENT_REASON_CODES Enumeration

Wi-Fi Soft AP event reason codes

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_SOFTAP_EVENT_LINK_LOST = 0,
  DRV_WIFI_SOFTAP_EVENT_RECEIVED_DEAUTH = 1
} DRV_WIFI_SOFT_AP_EVENT_REASON_CODES;
```

Description

Wi-Fi Soft AP event reason codes

This enumeration identifies Wi-Fi Soft AP events.

DRV_WIFI_SOFT_AP_STATES Enumeration

Wi-Fi Soft AP events

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_SOFTAP_EVENT_CONNECTED = 0,
  DRV_WIFI_SOFTAP_EVENT_DISCONNECTED = 1
} DRV_WIFI_SOFT_AP_STATES;
```

Description

Wi-Fi Soft AP events

This enumeration identifies Wi-Fi Soft AP events.

DRV_WIFI_STATUS_CODES Enumeration

Selection of different codes when Wi-Fi connection fails due to association or authentication failure.

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_UNSPECIFIED_FAILURE = 1,
  DRV_WIFI_CAPS_UNSUPPORTED = 10,
  DRV_WIFI_REASSOC_NO_ASSOC = 11,
  DRV_WIFI_ASSOC_DENIED_UNSPEC = 12,
  DRV_WIFI_NOT_SUPPORTED_AUTH_ALG = 13,
  DRV_WIFI_UNKNOWN_AUTH_TRANSACTION = 14,
  DRV_WIFI_CHALLENGE_FAIL = 15,
  DRV_WIFI_AUTH_TIMEOUT = 16,
  DRV_WIFI_AP_UNABLE_TO_HANDLE_NEW_STA = 17,
  DRV_WIFI_ASSOC_DENIED_RATES = 18,
  DRV_WIFI_ASSOC_DENIED_NOSHORTPREAMBLE = 19,
  DRV_WIFI_ASSOC_DENIED_NOPBCC = 20,
  DRV_WIFI_ASSOC_DENIED_NOAGILITY = 21,
  DRV_WIFI_ASSOC_DENIED_NOSHORTTIME = 25,
  DRV_WIFI_ASSOC_DENIED_NODSSSOFDM = 26,
  DRV_WIFI_S_INVALID_IE = 40,
  DRV_WIFI_S_INVALID_GROUPCIPHER = 41,
  DRV_WIFI_S_INVALID_PAIRWISE_CIPHER = 42,
  DRV_WIFI_S_INVALID_AKMP = 43,
  DRV_WIFI_UNSUPPORTED_RSN_VERSION = 44,
  DRV_WIFI_S_INVALID_RSN_IE_CAP = 45,
  DRV_WIFI_S_CIPHER_SUITE_REJECTED = 46,
  DRV_WIFI_TIMEOUT = 47
} DRV_WIFI_STATUS_CODES;
```

Description

Status codes for connection for association or authentication failure

This enumeration identifies the codes for a connection failure due to association or authentication failure. These codes are used in DRV_WIFI_ProcessEvent(), case DRV_WIFI_EVENT_CONNECTION_FAILED.

DRV_WIFI_SWMULTICAST_CONFIG Structure

Contains data pertaining to Wi-Fi software multicast filter configuration

File

drv_wifi.h

C

```
typedef struct {
  uint8_t filterId;
  uint8_t action;
  uint8_t macBytes[6];
  uint8_t macBitMask;
} DRV_WIFI_SWMULTICAST_CONFIG;
```

Members

Members	Description
uint8_t filterId;	DRV_WIFI_MULTICAST_FILTER_1 through DRV_WIFI_MULTICAST_FILTER_16
uint8_t action;	configures the multicast filter (see description)
uint8_t macBytes[6];	Array containing the MAC address to filter on (using the destination address of each incoming 802.11 frame). Specific bytes within the MAC address can be designated as "don't care" bytes. See macBitMask. This field in only used if action = WF_MULTICAST_USE_FILTERS.
uint8_t macBitMask;	A byte where bits 5:0 correspond to macBytes[5:0]. If the bit is zero then the corresponding MAC byte must be an exact match for the frame to be forwarded to the Host PIC. If the bit is one then the corresponding MAC byte is a "don't care" and not used in the Multicast filtering process. This field in only used if action = WF_MULTICAST_USE_FILTERS.

Description

Wi-Fi SW Multicast Filter Config

This structure contains data pertaining to the configuration of the software multicast config filter.

'action' Field	Description
DRV_WIFI_MULTICAST_DISABLE_ALL	Multicast filter discards all received multicast messages.
DRV_WIFI_MULTICAST_ENABLE_ALL	Multicast filter forwards all received multicast messages to host.
DRV_WIFI_MULTICAST_USE_FILTERS	The MAC filter will be used and the remaining fields define the filter.

DRV_WIFI_TX_MODES Enumeration

Selections for Wi-Fi TX mode

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_TXMODE_G_RATES = 0,
  DRV_WIFI_TXMODE_B_RATES = 1,
  DRV_WIFI_TXMODE_LEGACY_RATES = 2
} DRV_WIFI_TX_MODES;
```

Members

Members	Description
DRV_WIFI_TXMODE_G_RATES = 0	Use 802.11 'g' rates
DRV_WIFI_TXMODE_B_RATES = 1	Use only 802.11 'b' rates
DRV_WIFI_TXMODE_LEGACY_RATES = 2	Use only 1 and 2 mbps rates

Description

TX Modes

This enumeration identifies the choices the MRF24W TX mode. It is recommended to use the DRV_WIFI_TXMODE_G_RATES for best performance. See DRV_WIFI_TxModeSet().

DRV_WIFI_WEP_CONTEXT Structure

Contains data pertaining to Wi-Fi WEP context

File

drv_wifi.h

C

```
typedef struct {
  uint8_t wepSecurityType;
  uint8_t wepKey[DRV_WIFI_MAX_WEP_KEY_LENGTH];
  uint8_t wepKeyLength;
  uint8_t wepKeyType;
} DRV_WIFI_WEP_CONTEXT;
```

Members

Members	Description
uint8_t wepSecurityType;	DRV_WIFI_SECURITY_WEP_40 or DRV_WIFI_SECURITY_WEP_104
uint8_t wepKey[DRV_WIFI_MAX_WEP_KEY_LENGTH];	Array containing four WEP binary keys. This will be four, 5-byte keys for WEP-40 or four, thirteen-byte keys for WEP-104.
uint8_t wepKeyLength;	number of bytes pointed to by p_wepKey
uint8_t wepKeyType;	DRV_WIFI_SECURITY_WEP_OPENKEY (default) or DRV_WIFI_SECURITY_WEP_SHAREDKEY

Description

Wi-Fi Wep Security Context

This structure contains MRF24WG WEP context. See DRV_WIFI_SecurityWepSet().

DRV_WIFI_WEP_KEY_TYPE Enumeration

Selections for WEP key type when using WEP security.

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_SECURITY_WEP_SHAREDKEY = 0,
  DRV_WIFI_SECURITY_WEP_OPENKEY = 1
} DRV_WIFI_WEP_KEY_TYPE;
```

Members

Members	Description
DRV_WIFI_SECURITY_WEP_SHAREDKEY = 0	use WEP shared key
DRV_WIFI_SECURITY_WEP_OPENKEY = 1	use WEP open key (default)

Description

WEP Key Types

This enumeration identifies the choices for the WEP key type when using WEP security. The recommended key type (and default) is Open key.

DRV_WIFI_WPA_CONTEXT Structure

Contains data pertaining to Wi-Fi WPA.

File

drv_wifi.h

C

```
typedef struct {
  uint8_t wpaSecurityType;
  DRV_WIFI_WPA_KEY_INFO keyInfo;
} DRV_WIFI_WPA_CONTEXT;
```

Members

Members	Description
uint8_t wpaSecurityType;	desired security type (see description)
DRV_WIFI_WPA_KEY_INFO keyInfo;	see DRV_WIFI_WPA_KEY_INFO

Description

Wi-Fi WPA context

This structure contains MRF24WG WPA context. See DRV_WIFI_SecurityWpaSet().

DRV_WIFI_SECURITY_WPA_AUTO_WITH_KEY	Auto-select between WPA/WPA2 with binary key
DRV_WIFI_SECURITY_WPA_AUTO_WITH_PASS_PHRASE	Auto-select between WPA/WPA2 with passphrase

DRV_WIFI_WPA_KEY_INFO Structure

Contains data pertaining to Wi-Fi WPA Key

File

drv_wifi.h

C

```
typedef struct {
  uint8_t key[DRV_WIFI_MAX_WPA_PASS_PHRASE_LENGTH];
  uint8_t keyLength;
} DRV_WIFI_WPA_KEY_INFO;
```

Members

Members	Description
uint8_t key[DRV_WIFI_MAX_WPA_PASS_PHRASE_LENGTH];	binary key or passphrase
uint8_t keyLength;	number of bytes in binary key (always 32) or passphrase

Description

Wi-Fi WPA Key context

This structure contains MRF24WG WPA key info. This structure is used in the DRV_WIFI_WPA_CONTEXT and DRV_WIFI_WPS_CONTEXT structures.

DRV_WIFI_WPS_AUTH_TYPES Enumeration

Selection of WPS Authorization types

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_WPS_AUTH_OPEN = 0x01,
  DRV_WIFI_WPS_AUTH_WPA_PSK = 0x02,
  DRV_WIFI_WPS_AUTH_SHARED = 0x04,
  DRV_WIFI_WPS_AUTH_WPA = 0x08,
  DRV_WIFI_WPS_AUTH_WPA2 = 0x10,
  DRV_WIFI_WPS_AUTH_WPA2_PSK = 0x20
} DRV_WIFI_WPS_AUTH_TYPES;
```

Description

Wi-Fi WPS authorization types

This enumeration identifies the WPS authorization types

DRV_WIFI_WPS_CONTEXT Structure

Contains data pertaining to Wi-Fi WPS security.

File

drv_wifi.h

C

```
typedef struct {
  uint8_t wpsSecurityType;
  uint8_t wpsPin[DRV_WIFI_WPS_PIN_LENGTH];
  uint8_t wpsPinLength;
} DRV_WIFI_WPS_CONTEXT;
```

Members

Members	Description
uint8_t wpsSecurityType;	DRV_WIFI_SECURITY_WPS_PUSH_BUTTON or DRV_WIFI_SECURITY_WPS_PIN
uint8_t wpsPin[DRV_WIFI_WPS_PIN_LENGTH];	if using DRV_WIFI_SECURITY_WPS_PIN then pointer to 8-digit pin
uint8_t wpsPinLength;	should always be 8 if used, 0 if not used

Description

Wi-Fi WPS security context

This structure contains MRF24WG WPS security context. See DRV_WIFI_SecurityWpsSet().

DRV_WIFI_WPS_CREDENTIAL Structure

Contains data pertaining to Wi-Fi WPS Credentials

File

drv_wifi.h

C

```
typedef struct {
  uint8_t ssid[DRV_WIFI_MAX_SSID_LENGTH];
  uint8_t netKey[DRV_WIFI_MAX_SECURITY_KEY_LENGTH];
  uint16_t authType;
  uint16_t encType;
  uint8_t netIdx;
  uint8_t ssidLen;
  uint8_t keyIdx;
  uint8_t keyLen;
  uint8_t bssid[DRV_WIFI_BSSID_LENGTH];
} DRV_WIFI_WPS_CREDENTIAL;
```

Members

Members	Description
uint8_t ssid[DRV_WIFI_MAX_SSID_LENGTH];	network SSID
uint8_t netKey[DRV_WIFI_MAX_SECURITY_KEY_LENGTH];	binary security key (not used if security is open)
uint16_t authType;	WPS authorization type (see description)
uint16_t encType;	Encoding type (see description)
uint8_t netIdx;	not used
uint8_t ssidLen;	number of bytes in SSID
uint8_t keyIdx;	Only valid encType = WF_ENC_WEP. This is the index of the WEP key being used.
uint8_t keyLen;	number of bytes in netKey
uint8_t bssid[DRV_WIFI_BSSID_LENGTH];	MAC address of AP

Description

Wi-Fi WPS Credentials

This structure contains data pertaining to the configuration of the Wi-Fi WPS credentials.

'authType' Field	Description
DRV_WIFI_WPS_AUTH_OPEN	open security
DRV_WIFI_WPS_AUTH_WPA_PSK	WPA with PSK
DRV_WIFI_WPS_AUTH_SHARED	Shared key
DRV_WIFI_WPS_AUTH_WPA	WPA
DRV_WIFI_WPS_AUTH_WPA2	WPA2
DRV_WIFI_WPS_AUTH_WPA2_PSK	WPA2 with PSK

'encType' Field	Description
DRV_WIFI_WPS_ENC_NONE	No encoding
DRV_WIFI_WPS_ENC_WEP	WEP encoding

DRV_WIFI_WPS_ENC_TKIP	WPS/TKIP encodingShared key
DRV_WIFI_ENC_AES	AES encoding

DRV_WIFI_WPS_ENCODE_TYPES Enumeration

Selection of WPS Encoding types

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_WPS_ENC_NONE = 0x01,
  DRV_WIFI_WPS_ENC_WEP = 0x02,
  DRV_WIFI_WPS_ENC_TKIP = 0x04,
  DRV_WIFI_ENC_AES = 0x08
} DRV_WIFI_WPS_ENCODE_TYPES;
```

Description

Wi-Fi WPS encoding types

This enumeration identifies the WPS encoding types

ENABLE_P2P_PRINTS Macro

File

drv_wifi.h

C

```
#define ENABLE_P2P_PRINTS ((uint8_t)(1 << 1)) /* not supported */
```

Description

not supported

© 2013-2015 Microchip Technology Inc.

ENABLE_WPS_PRINTS Macro

File

drv_wifi.h

C

```
#define ENABLE_WPS_PRINTS ((uint8_t)(1 << 0))
```

Description

This is macro ENABLE_WPS_PRINTS.

© 2013-2015 Microchip Technology Inc.

WF_WPS_PIN_LENGTH Macro

File

drv_wifi.h

C

```
#define WF_WPS_PIN_LENGTH 8
```

Description

WPS PIN Length

DRV_WIFI_P2P_ERROR_CODES Enumeration

Selection of different codes during a P2P connection.

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_WFD_SUCCESS = 0,
  DRV_WIFI_WFD_INFO_CURRENTLY_UNAVAILABLE = 1,
  DRV_WIFI_WFD_INCOMPATIBLE_PARAMS = 2,
  DRV_WIFI_WFD_LIMIT_REACHED = 3,
  DRV_WIFI_WFD_INVALID_PARAMS = 4,
  DRV_WIFI_WFD_UNABLE_TO_ACCOMMODATE = 5,
  DRV_WIFI_WFD_PREV_PROTOCOL_ERROR = 6,
  DRV_WIFI_WFD_NO_COMMON_CHANNELS = 7,
  DRV_WIFI_WFD_UNKNOWN_GROUP = 8,
  DRV_WIFI_WFD_INCOMPATIBLE_PROV_METHOD = 10,
  DRV_WIFI_WFD_REJECTED_BY_USER = 11,
  DRV_WIFI_WFD_NO_MEM = 12,
  DRV_WIFI_WFD_INVALID_ACTION = 13,
  DRV_WIFI_WFD_TX_FAILURE = 14,
  DRV_WIFI_WFD_TIME_OUT = 15
} DRV_WIFI_P2P_ERROR_CODES;
```

Description

P2P Error codes

This enumeration identifies the error codes that can take place during a P2P connection.

DRV_WIFI_P2P_STATES Enumeration

Selection of different states during a P2P connection.

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_P2P_IDLE = 0,
  DRV_WIFI_P2P_SCAN = 1,
  DRV_WIFI_P2P_LISTEN = 2,
  DRV_WIFI_P2P_FIND = 3,
  DRV_WIFI_P2P_START_FORMATION = 4,
  DRV_WIFI_P2P_NEG_REQ_DONE = 5,
  DRV_WIFI_P2P_WAIT_NEG_REQ_DONE = 6,
  DRV_WIFI_P2P_WAIT_FORMATION_DONE = 7,
  DRV_WIFI_P2P_INVITE = 8,
  DRV_WIFI_P2P_PROVISION = 9,
  DRV_WIFI_P2P_CLIENT = 10
} DRV_WIFI_P2P_STATES;
```

Description

P2P State codes

This enumeration identifies the codes that can take place during a P2P connection.

DRV_WIFI_WPS_ERROR_CONFIG_CODES Enumeration

Selection of different codes when a WPS connection fails.

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_WPS_NOERR = 0,
  DRV_WIFI_WPS_SESSION_OVERLAPPED = 1,
  DRV_WIFI_WPS_DECRYPT_CRC_FAILURE = 2,
  DRV_WIFI_WPS_24G_NOT_SUPPORTED = 3,
  DRV_WIFI_WPS_RETRY_FAILURE = 4,
  DRV_WIFI_WPS_INVALID_MSG = 5,
  DRV_WIFI_WPS_AUTH_FAILURE = 6,
  DRV_WIFI_WPS_ASSOC_FAILURE = 7,
  DRV_WIFI_WPS_MSG_TIMEOUT = 8,
  DRV_WIFI_WPS_SESSION_TIMEOUT = 9,
  DRV_WIFI_WPS_DEVPASSWD_AUTH_FAILURE = 10,
  DRV_WIFI_WPS_NO_CONN_TOREG = 11,
  DRV_WIFI_WPS_MULTI_PBC_DETECTED = 12,
  DRV_WIFI_WPS_EAP_FAILURE = 13,
  DRV_WIFI_WPS_DEV_BUSY = 14,
  DRV_WIFI_WPS_SETUP_LOCKED = 15
} DRV_WIFI_WPS_ERROR_CONFIG_CODES;
```

Description

WPS Config Error Codes

This enumeration identifies the codes that can take place when WPS fails.

DRV_WIFI_WPS_STATE_CODES Enumeration

Selection of different codes when a Extensible Authentication Protocol is used.

File

drv_wifi.h

C

```
typedef enum {
  DRV_WIFI_EAPOL_START = 1,
  DRV_WIFI_EAP_REQ_IDENTITY = 2,
  DRV_WIFI_EAP_RSP_IDENTITY = 3,
  DRV_WIFI_EAP_WPS_START = 4,
  DRV_WIFI_EAP_RSP_M1 = 5,
  DRV_WIFI_EAP_REQ_M2 = 6,
  DRV_WIFI_EAP_RSP_M3 = 7,
  DRV_WIFI_EAP_REQ_M4 = 8,
  DRV_WIFI_EAP_RSP_M5 = 9,
  DRV_WIFI_EAP_REQ_M6 = 10,
  DRV_WIFI_EAP_RSP_M7 = 11,
  DRV_WIFI_EAP_REQ_M8 = 12,
  DRV_WIFI_EAP_RSP_DONE = 13,
  DRV_WIFI_EAP_FAILURE = 14
} DRV_WIFI_WPS_STATE_CODES;
```

Description

WPS State codes

This enumeration identifies the codes that can take place when using EAPOL.

DRV_GFX_SSD1926_COMMAND Structure

Structure for the commands in the driver queue.

File

drv_gfx_ssd1926.h

C

```
typedef struct {
  uint32_t address;
  uint16_t * array;
  uint16_t data;
} DRV_GFX_SSD1926_COMMAND;
```

Members

Members	Description
uint32_t address;	whether or not the task is complete

Description

Structure: DRV_GFX_SSD1926_COMMAND

Structure for the commands in the driver queue.

Parameters

Parameters	Description
address	pixel address
array	pointer to array of pixel data
data	pixel color

DRV_WIFI_DEFAULT_WEP_KEY_INDEX Macro

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_WEP_KEY_INDEX 0
```

Description

see DRV_WIFI_SecurityWepSet() and DRV_WIFI_WEP_CONTEXT

DRV_WIFI_SOFTAP_NETWORK_CONTEXT Structure

Contains data pertaining to Wi-Fi SoftAP context

File

drv_wifi.h

C

```
typedef struct {
  bool hiddenSsid;
} DRV_WIFI_SOFTAP_NETWORK_CONTEXT;
```

Members

Members	Description
bool hiddenSsid;	When starting an SoftAP network, the SSID can be hidden in the beacons. Set true to hide the SSID, else false. Default is false.

Description

Wi-Fi SoftAP Context

This structure contains MRF24WG SoftAP context data. See DRV_WIFI_SoftAPContextSet().

DRV_WIFI_DEFAULT_SOFTAP_HIDDEN_SSID Macro

Default values for Wi-Fi SoftAP settings

File

drv_wifi.h

C

```
#define DRV_WIFI_DEFAULT_SOFTAP_HIDDEN_SSID false
```

Description

Wi-Fi SoftAP default settings

These defines identify various default Wi-Fi SoftAP settings that can be used in the DRV_WIFI_SOFTAP_NETWORK_CONTEXT structure.

Files

Files

Name	Description
drv_wifi.h	Wi-Fi specific MAC function prototypes called by TCP/IP stack.

Description

This section lists the source and header files used by the MRF24W Wi-Fi Driver Library.

drv_wifi.h

Wi-Fi specific MAC function prototypes called by TCP/IP stack.

Enumerations

	Name	Description
	adhocMode	Selection of different AdHoc connection modes
	DRV_WIFI_ADHOC_MODES	Selection of different AdHoc connection modes
	DRV_WIFI_CONNECTION_STATES	Wi-Fi Connection states
	DRV_WIFI_DEVICE_TYPES	Codes for Wi-Fi device types
	DRV_WIFI_DOMAIN_CODES	Regional domain codes.
	DRV_WIFI_EVENT_CONN_TEMP_LOST_CODES	Selection of different codes when Wi-Fi connection is temporarily lost.
	DRV_WIFI_EVENT_INFO	Selection of different EventInfo types
	DRV_WIFI_EVENTS	Selections for events that can occur.
	DRV_WIFI_GENERAL_ERRORS	This is type DRV_WIFI_GENERAL_ERRORS.
	DRV_WIFI_HIBERNATE_STATES	Wi-Fi Hibernate states
	DRV_WIFI_MGMT_ERRORS	Error codes returned when a management message is sent to the MRF24W
	DRV_WIFI_MULTICAST_FILTER_IDS	Selections for software Multicast filter IDs
	DRV_WIFI_MULTICAST_FILTERS	Selections for Software Multicast Filters.
	DRV_WIFI_P2P_ERROR_CODES	Selection of different codes during a P2P connection.
	DRV_WIFI_P2P_STATES	Selection of different states during a P2P connection.
	DRV_WIFI_POWER_SAVE_STATES	Wi-Fi Power-Saving states
	DRV_WIFI_REASON_CODES	Selection of different codes when a deauthorization or disassociation event has occurred.
	DRV_WIFI_RECONNECT_MODES	Selection of different Reconnection modes
	DRV_WIFI_SCAN_TYPES	Selection of different Wi-Fi scan types
	DRV_WIFI_SOFT_AP_EVENT_REASON_CODES	Wi-Fi Soft AP event reason codes
	DRV_WIFI_SOFT_AP_STATES	Wi-Fi Soft AP events
	DRV_WIFI_STATUS_CODES	Selection of different codes when Wi-Fi connection fails due to association or authentication failure.
	DRV_WIFI_TX_MODES	Selections for Wi-Fi TX mode
	DRV_WIFI_WEP_KEY_TYPE	Selections for WEP key type when using WEP security.
	DRV_WIFI_WPS_AUTH_TYPES	Selection of WPS Authorization types
	DRV_WIFI_WPS_ENCODE_TYPES	Selection of WPS Encoding types

	DRV_WIFI_WPS_ERROR_CONFIG_CODES	Selection of different codes when a WPS connection fails.
	DRV_WIFI_WPS_STATE_CODES	Selection of different codes when a Extensible Authentication Protocol is used.

Functions

	Name	Description
	DRV_WIFI_AdhocContextSet	Sets the AdHoc context. **Implementation:** Dynamic
	DRV_WIFI_BssidGet	Gets the BSSID set in DRV_WIFI_BssidSet(). **Implementation:** Dynamic
	DRV_WIFI_BssidSet	Sets the Basic Service Set Identifier (BSSID). **Implementation:** Dynamic
	DRV_WIFI_ChannelListGet	Gets the channel list. **Implementation:** Dynamic
	DRV_WIFI_ChannelListSet	Sets the channel list. **Implementation:** Dynamic
	DRV_WIFI_ConfigDataErase	Erases configuration data from the board EEPROM. **Implementation:** Dynamic
	DRV_WIFI_ConfigDataLoad	Loads configuration data from the board EEPROM. **Implementation:** Dynamic
	DRV_WIFI_ConfigDataPrint	Outputs to console the configuration data from the board EEPROM. **Implementation:** Dynamic
	DRV_WIFI_ConfigDataSave	Save configuration data to the board EEPROM. **Implementation:** Dynamic
	DRV_WIFI_Connect	Directs the MRF24WG to connect to a Wi-Fi network. **Implementation:** Dynamic
	DRV_WIFI_ConnectContextGet	Gets the current Wi-Fi connection context. **Implementation:** Dynamic
	DRV_WIFI_ConnectionStateGet	Gets the current Wi-Fi connection state. **Implementation:** Dynamic
	DRV_WIFI_ConnectStateMachine	Starts the state machine to connect to an 802.11 network. **Implementation:** Dynamic
	DRV_WIFI_Deinitialize	Initializes the MRF24WG Wi-Fi driver. **Implementation:** Dynamic
	DRV_WIFI_DeviceInfoGet	Retrieves MRF24WG device information. **Implementation:** Dynamic
	DRV_WIFI_Disconnect	Directs the MRF24WG to disconnect from a Wi-Fi network. **Implementation:** Dynamic
	DRV_WIFI_EasyConfigTask_RtosTask	Implements Wi-Fi driver easy configuration RTOS task. **Implementation:** Dynamic
	DRV_WIFI_GratuitousArpStart	Starts a periodic gratuitous ARP response. **Implementation:** Dynamic
	DRV_WIFI_GratuitousArpStop	Stops a periodic gratuitous ARP. **Implementation:** Dynamic
	DRV_WIFI_HibernateEnable	Puts the MRF24WG into hibernate mode. **Implementation:** Dynamic

	DRV_WIFI_HWMulticastFilterGet	Gets a multicast address filter from one of the two multicast filters. **Implementation:** Dynamic
	DRV_WIFI_HWMulticastFilterSet	Sets a multicast address filter using one of the two hardware multicast filters. **Implementation:** Dynamic
	DRV_WIFI_Initialize	Initializes the MRF24WG Wi-Fi driver. **Implementation:** Dynamic
	DRV_WIFI_InitStateMachine_RtosTask	Implements Wi-Fi driver initialization RTOS task. **Implementation:** Dynamic
	DRV_WIFI_INT_Handle	Wi-Fi driver interrupt handle. **Implementation:** Dynamic
	DRV_WIFI_isHibernateEnable	Checks if MRF24W is in hibernate mode. **Implementation:** Dynamic
	DRV_WIFI_ISR_RtosTask	Implements Wi-Fi driver ISR RTOS task. **Implementation:** Dynamic
	DRV_WIFI_ISR_SemLock	Locks semaphore in Wi-Fi RTOS ISR. **Implementation:** Dynamic
	DRV_WIFI_ISR_SemUnlock	Unlocks semaphore in Wi-Fi RTOS ISR. **Implementation:** Dynamic
	DRV_WIFI_MacAddressGet	Retrieves the MRF24WG MAC address. **Implementation:** Dynamic
	DRV_WIFI_MacAddressSet	Uses a different MAC address for the MRF24W. **Implementation:** Dynamic
	DRV_WIFI_MACProcess_RtosTask	Implements Wi-Fi driver MAC process RTOS task. **Implementation:** Dynamic
	DRV_WIFI_MacStatsGet	Gets MAC statistics. **Implementation:** Dynamic
	DRV_WIFI_MRF24W_ISR	Wi-Fi driver (MRF24WG specific) interrupt service routine. **Implementation:** Dynamic
	DRV_WIFI_NetworkTypeGet	Gets the Wi-Fi network type. **Implementation:** Dynamic
	DRV_WIFI_NetworkTypeSet	Sets the Wi-Fi network type. **Implementation:** Dynamic
	DRV_WIFI_PowerSaveStateGet	Gets the current power-saving state.
	DRV_WIFI_ProcessEvent	Processes Wi-Fi event. **Implementation:** Dynamic
	DRV_WIFI_PsPollDisable	Disables PS-Poll mode. **Implementation:** Dynamic
	DRV_WIFI_PsPollEnable	Enables PS Poll mode. **Implementation:** Dynamic
	DRV_WIFI_ReconnectModeGet	Gets the Wi-Fi reconnection mode. **Implementation:** Dynamic
	DRV_WIFI_ReconnectModeSet	Sets the Wi-Fi reconnection mode. **Implementation:** Dynamic
	DRV_WIFI_RegionalDomainGet	Retrieves the MRF24WG Regional domain. **Implementation:** Dynamic
	DRV_WIFI_RSSI_Cache_FromRxDataRead	Caches RSSI value from Rx data packet. **Implementation:** Dynamic

	Name	Description
	DRV_WIFI_RSSI_Get_FromRxDataRead	Reads RSSI value from Rx data packet. **Implementation:** Dynamic
	DRV_WIFI_RssiGet	Gets RSSI value set in DRV_WIFI_RssiSet(). **Implementation:** Dynamic
	DRV_WIFI_RssiSet	Sets RSSI restrictions when connecting. **Implementation:** Dynamic
	DRV_WIFI_RTOS_TaskInit	Initializes RTOS tasks for Wi-Fi driver. **Implementation:** Dynamic
	DRV_WIFI_RtsThresholdGet	Gets the RTS Threshold. **Implementation:** Dynamic
	DRV_WIFI_RtsThresholdSet	Sets the RTS Threshold. **Implementation:** Dynamic
	DRV_WIFI_Scan	Commands the MRF24W to start a scan operation. This will generate the WF_EVENT_SCAN_RESULTS_READY event. **Implementation:** Dynamic
	DRV_WIFI_ScanContextGet	Gets the Wi-Fi scan context. **Implementation:** Dynamic
	DRV_WIFI_ScanContextSet	Sets the Wi-Fi scan context. **Implementation:** Dynamic
	DRV_WIFI_ScanGetResult	Read selected scan results back from MRF24W. **Implementation:** Dynamic
	DRV_WIFI_SecurityGet	Gets the current Wi-Fi security setting. **Implementation:** Dynamic
	DRV_WIFI_SecurityOpenSet	Sets Wi-Fi security to open (no security). **Implementation:** Dynamic
	DRV_WIFI_SecurityTypeGet	This is function DRV_WIFI_SecurityTypeGet.
	DRV_WIFI_SecurityWepSet	Sets Wi-Fi security to use WEP. **Implementation:** Dynamic
	DRV_WIFI_SecurityWpaSet	Sets Wi-Fi security to use WPA or WPA2. **Implementation:** Dynamic
	DRV_WIFI_SecurityWpsSet	Sets Wi-Fi security to use WPS. **Implementation:** Dynamic
	DRV_WIFI_SetLinkDownThreshold	Sets number of consecutive Wi-Fi TX failures before link is considered down. **Implementation:** Dynamic
	DRV_WIFI_SetPSK	Sets the binary WPA PSK code in WPS. **Implementation:** Dynamic
	DRV_WIFI_SoftAPContextSet	Sets the SoftAP context. **Implementation:** Dynamic
	DRV_WIFI_SoftApEventInfoGet	Gets the stored Soft AP event info. **Implementation:** Dynamic
	DRV_WIFI_SpiClose	Closes SPI object for Wi-Fi driver. **Implementation:** Dynamic
	DRV_WIFI_SpiInit	Initializes SPI object for Wi-Fi driver. **Implementation:** Dynamic
	DRV_WIFI_SsidGet	Gets the SSID. **Implementation:** Dynamic

	DRV_WIFI_SsidSet	Sets the SSID. **Implementation:** Dynamic
	DRV_WIFI_SWMultiCastFilterEnable	Forces the MRF24WG to use software multicast filters instead of hardware multicast filters. **Implementation:** Dynamic
	DRV_WIFI_SWMulticastFilterSet	Sets a multicast address filter using one of the software multicast filters. **Implementation:** Dynamic
	DRV_WIFI_TASK_MUTEX_Lock	Locks MUTEX in Wi-Fi RTOS task when necessary. **Implementation:** Dynamic
	DRV_WIFI_TASK_MUTEX_Unlock	Unlocks MUTEX in Wi-Fi RTOS task. **Implementation:** Dynamic
	DRV_WIFI_TxModeGet	Gets 802.11 TX mode. **Implementation:** Dynamic
	DRV_WIFI_TxModeSet	Configures 802.11 TX mode. **Implementation:** Dynamic
	DRV_WIFI_TxPowerFactoryMaxGet	Retrieves the factory-set max TX power from the MRF24W. **Implementation:** Dynamic
	DRV_WIFI_TxPowerMaxGet	Gets the TX max power on the MRF24WG0M. **Implementation:** Dynamic
	DRV_WIFI_TxPowerMaxSet	Sets the TX max power on the MRF24WG0M. **Implementation:** Dynamic
	DRV_WIFI_WepKeyTypeGet	Gets the WEP Key type. **Implementation:** Dynamic
	DRV_WIFI_WPSCredentialsGet	Gets the WPS credentials. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_WIFI_BSSID_LENGTH	This is macro DRV_WIFI_BSSID_LENGTH.
	DRV_WIFI_DEAUTH_REASONCODE_MASK	This is macro DRV_WIFI_DEAUTH_REASONCODE_MASK.
	DRV_WIFI_DEFAULT_ADHOC_BEACON_PERIOD	ms
	DRV_WIFI_DEFAULT_ADHOC_HIDDEN_SSID	Default values for Wi-Fi AdHoc settings
	DRV_WIFI_DEFAULT_ADHOC_MODE	This is macro DRV_WIFI_DEFAULT_ADHOC_MODE.
	DRV_WIFI_DEFAULT_PS_DTIM_ENABLED	DTIM wake-up enabled (normally the case)
	DRV_WIFI_DEFAULT_PS_DTIM_INTERVAL	number of beacon periods
	DRV_WIFI_DEFAULT_PS_LISTEN_INTERVAL	100ms multiplier, e.g. 1 * 100ms = 100ms
	DRV_WIFI_DEFAULT_SCAN_COUNT	Default values for Wi-Fi scan context
	DRV_WIFI_DEFAULT_SCAN_MAX_CHANNEL_TIME	ms
	DRV_WIFI_DEFAULT_SCAN_MIN_CHANNEL_TIME	ms
	DRV_WIFI_DEFAULT_SCAN_PROBE_DELAY	us
	DRV_WIFI_DEFAULT_SOFTAP_HIDDEN_SSID	Default values for Wi-Fi SoftAP settings
	DRV_WIFI_DEFAULT_WEP_KEY_INDEX	see DRV_WIFI_SecurityWepSet() and DRV_WIFI_WEP_CONTEXT

	DRV_WIFI_DEFAULT_WEP_KEY_TYPE	This is macro DRV_WIFI_DEFAULT_WEP_KEY_TYPE .
	DRV_WIFI_DISABLED	
	DRV_WIFI_DISASSOC_REASONCODE_MASK	This is macro DRV_WIFI_DISASSOC_REASONCODE_MASK.
	DRV_WIFI_ENABLED	This is macro DRV_WIFI_ENABLED.
	DRV_WIFI_MAX_CHANNEL_LIST_LENGTH	This is macro DRV_WIFI_MAX_CHANNEL_LIST_LENGTH.
	DRV_WIFI_MAX_NUM_RATES	This is macro DRV_WIFI_MAX_NUM_RATES.
	DRV_WIFI_MAX_SECURITY_KEY_LENGTH	This is macro DRV_WIFI_MAX_SECURITY_KEY_LENGTH.
	DRV_WIFI_MAX_SSID_LENGTH	This is macro DRV_WIFI_MAX_SSID_LENGTH.
	DRV_WIFI_MAX_WEP_KEY_LENGTH	This is macro DRV_WIFI_MAX_WEP_KEY_LENGTH.
	DRV_WIFI_MAX_WPA_PASS_PHRASE_LENGTH	must exclude string terminator
	DRV_WIFI_MIN_WPA_PASS_PHRASE_LENGTH	must exclude string terminator
	DRV_WIFI_NETWORK_TYPE_ADHOC	This is macro DRV_WIFI_NETWORK_TYPE_ADHOC.
	DRV_WIFI_NETWORK_TYPE_INFRASTRUCTURE	Selection of different Wi-Fi network types
	DRV_WIFI_NETWORK_TYPE_P2P	not supported
	DRV_WIFI_NETWORK_TYPE_SOFT_AP	This is macro DRV_WIFI_NETWORK_TYPE_SOFT_AP.
	DRV_WIFI_NO_ADDITIONAL_INFO	eventInfo define for DRV_WIFI_ProcessEvent() when no additional info is supplied
	DRV_WIFI_RETRY_ADHOC	This is macro DRV_WIFI_RETRY_ADHOC.
	DRV_WIFI_RETRY_FOREVER	This is macro DRV_WIFI_RETRY_FOREVER.
	DRV_WIFI_RTS_THRESHOLD_MAX	maximum RTS threshold size in bytes
	DRV_WIFI_SECURITY_EAP	not supported
	DRV_WIFI_SECURITY_OPEN	Selection of different Wi-Fi security types
	DRV_WIFI_SECURITY_WEP_104	This is macro DRV_WIFI_SECURITY_WEP_104.
	DRV_WIFI_SECURITY_WEP_40	This is macro DRV_WIFI_SECURITY_WEP_40.
	DRV_WIFI_SECURITY_WPA_AUTO_WITH_KEY	This is macro DRV_WIFI_SECURITY_WPA_AUTO_WITH_KEY.
	DRV_WIFI_SECURITY_WPA_AUTO_WITH_PASS_PHRASE	This is macro DRV_WIFI_SECURITY_WPA_AUTO_WITH_PASS_PHRASE.
	DRV_WIFI_SECURITY_WPS_PIN	This is macro DRV_WIFI_SECURITY_WPS_PIN.

	DRV_WIFI_SECURITY_WPS_PUSH_BUTTON	This is macro DRV_WIFI_SECURITY_WPS_PUSH_BUTTON.
	DRV_WIFI_WEP104_KEY_LENGTH	4 keys of 13 bytes each
	DRV_WIFI_WEP40_KEY_LENGTH	4 keys of 5 bytes each
	DRV_WIFI_WPA_KEY_LENGTH	This is macro DRV_WIFI_WPA_KEY_LENGTH.
	DRV_WIFI_WPS_PIN_LENGTH	7 digits + checksum byte
	ENABLE_P2P_PRINTS	not supported
	ENABLE_WPS_PRINTS	This is macro ENABLE_WPS_PRINTS.
	WF_WPS_PIN_LENGTH	WPS PIN Length

Structures

	Name	Description
	DRV_WIFI_ADHOC_NETWORK_CONTEXT	Contains data pertaining to Wi-Fi AdHoc context
	DRV_WIFI_CONNECTION_CONTEXT	Contains data pertaining to MRF24WG connection context
	DRV_WIFI_DEVICE_INFO	Contains data pertaining to MRF24WG device type and version number
	DRV_WIFI_MAC_STATS	Wi-Fi MIB states
	DRV_WIFI_MGMT_INDICATE_SOFT_AP_EVENT	Contains data pertaining to Wi-Fi Soft AP event
	DRV_WIFI_PS_POLL_CONTEXT	Contains data pertaining to Wi-Fi PS-Poll context
	DRV_WIFI_SCAN_CONTEXT	Contains data pertaining to Wi-Fi scan context
	DRV_WIFI_SCAN_RESULT	Contains data pertaining to Wi-Fi scan results
	DRV_WIFI_SOFTAP_NETWORK_CONTEXT	Contains data pertaining to Wi-Fi SoftAP context
	DRV_WIFI_SWMULTICAST_CONFIG	Contains data pertaining to Wi-Fi software multicast filter configuration
	DRV_WIFI_WEP_CONTEXT	Contains data pertaining to Wi-Fi WEP context
	DRV_WIFI_WPA_CONTEXT	Contains data pertaining to Wi-Fi WPA.
	DRV_WIFI_WPA_KEY_INFO	Contains data pertaining to Wi-Fi WPA Key
	DRV_WIFI_WPS_CONTEXT	Contains data pertaining to Wi-Fi WPS security.
	DRV_WIFI_WPS_CREDENTIAL	Contains data pertaining to Wi-Fi WPS Credentials

Unions

	Name	Description
	DRV_WIFI_SECURITY_CONTEXT	Contains data pertaining to Wi-Fi security.

Description

Wi-Fi MAC interface functions

The functions in this header file are accessed by the TCP/IP stack via function pointers.

File Name

drv_wifi.h

Company

Microchip Technology Inc.

Index

_

_DRV_ADC_CONFIG_TEMPLATE_H macro 39
_DRV_AK4642_H macro 245
_DRV_AK4953_H macro 351
_DRV_COMMON_H macro 16
_DRV_ENCX24J600_Configuration structure 420
_DRV_I2C_INIT structure 719
_DRV_I2S_DATA16 structure 816
_DRV_I2S_DATA24 structure 817
_DRV_I2S_DATA32 structure 818
_DRV_MTCH6301_CLIENT_OBJECT structure 872
_DRV_PMP_QUEUE_ELEMENT_OBJ structure 996
_DRV_SDCARD_INIT structure 1049
_DRV_SPI_CLIENT_DATA structure 1135
_DRV_SPI_INIT structure 1125
_GFX_GFX_INIT structure 543
_PLIB_UNSUPPORTED macro 17
_QUEUE_ELEMENT_OBJECT structure 1001

A

Abstraction Model 22, 97, 139, 205, 257, 310, 389, 427, 465, 659, 745, 848, 896, 957, 1018, 1066, 1141, 1253, 1345, 1438, 1482
- ADC Driver Library 22
- AK4384 Codec Driver Library 139
- AK4642 Codec Driver Library 205
- AK4645 Codec Driver Library 257
- AK4953 Codec Driver Library 310
- Ethernet MAC Driver Library 427
- Ethernet PHY Driver Library 465
- MRF24W Wi-Fi Driver Library 1482
- MTCH6301 Driver Library 848
- NVM Driver Library 896
- PMP Driver Library 957
- SD Card Driver Library 1018
- SPI Driver Library 1066
- SPI Flash Driver Library 1141
- Timer Driver Library 1253
- USART Driver Library 1345
- USB Driver Library 1438

ADC Driver Library 20
adhocMode enumeration 1626
AK4384 Codec Driver Library 137
AK4642 Codec Driver Library 204
AK4645 Codec Driver Library 256
AK4953 Codec Driver Library 308
Alarm Functionality 1261

B

Block Operations 1145
Buffer Queue Transfer Model 1357
Building the Library 40, 100, 157, 220, 272, 321, 368, 393, 433, 470, 538, 673, 774, 858, 909, 967, 1026, 1089, 1167, 1269, 1370, 1464, 1491
- ADC Driver Library 40
- AK4384 Driver Library 157
- AK4642 Driver Library 220
- AK4645 Driver Library 272
- AK4953 Driver Library 321
- CPLD XC2C64A Driver Library 368
- ENCX24J600 Driver Library 393
- Ethernet MAC Driver Library 433
- Ethernet PHY Driver Library 470
- Graphics Driver Library 538
- I2C Driver Library 673
- I2S Driver Library 774
- MRF24W Wi-Fi Driver Library 1491
- MTCH6301 Driver Library 858
- NVM Driver Library 909
- PMP Driver Library 967
- SD Card Driver Library 1026
- SPI Driver Library 1089
- SPI Flash Driver Library 1167
- Timer Driver Library 1269
- USART Driver Library 1370
- USB Driver Library 1464

Bulk and Interrupt Transfers 1457
Byte Data Transfer Model 1354

C

Camera Driver Libraries 78
CAMERA_MODULE_ID enumeration 93
CAN Driver Library 127
Client Access 142, 208, 260, 312, 663, 748, 1069
Client Access Operation 899, 1021
Client Block Data Operation 900, 1022
Client Core Functionality 25
Client Functionality 1485
Client Interaction 1257
Client Operation 961
Client Operations 143, 208, 260, 312
Client Operations - Buffered 749
Client Operations - Non-buffered 756
Client Transfer 664
Client Transfer - Core 1070
Code Examples 26
Codec Driver Libraries 136
Comparator Driver Library 360
Configuring the Display 537
Configuring the Library 31, 98, 149, 214, 266, 314, 367, 392, 430, 467, 528, 667, 760, 854, 903, 965, 1024, 1073, 1147, 1264, 1361, 1462, 1490
- ADC Driver Library 31
- AK4384 Driver Library 149
- AK4642 Driver Library 214
- AK4645 Driver Library 266
- AK4953 Driver Library 314
- CPLD XC2C64A Driver Library 367
- Ethernet MAC Driver Library 430
- Ethernet PHY Driver Library 467
- Graphics Driver Library 528
- MRF24W Wi-Fi Driver Library 1490
- MTCH6301 Driver Library 854
- NVM Driver Library 903
- PMP Driver Library 965
- SD Card Driver Library 1024
- SPI Driver Library 1073

SPI Flash Driver Library 1147
Timer Driver Library 1264
USART Driver Library 1361
USB Driver Library 1462
Configuring the SPI Driver 391
Control Transfers 1452
Core Functionality 1260
Counter Modification 1259
CPLD XC2C64A Driver Library 364
CPLD_DEVICE_CONFIGURATION enumeration 378
CPLD_GFX_CONFIGURATION enumeration 379
CPLD_SPI_CONFIGURATION enumeration 380
CPLDGetDeviceConfiguration function 370
CPLDGetGraphicsConfiguration function 371
CPLDGetSPIConfiguration function 372
CPLDInitialize function 373
CPLDSetGraphicsConfiguration function 374
CPLDSetSPIFlashConfiguration function 375
CPLDSetWiFiConfiguration function 376
CPLDSetZigBeeConfiguration function 377
Creating a New Graphics Driver 526

D

DMA_ISR_TASK enumeration 565
Driver Initialization 1440
Driver Library Overview 2
Driver Tasks Routine 1359
driver.h 18
driver_common.h 18
drv_adc.h 75
DRV_ADC_ACQUISITION_TIME macro 31
DRV_ADC_ALTERNATE_INPUT_SAMPLING_ENABLE macro 32
DRV_ADC_ANALOG_INPUT macro 32
DRV_ADC_AUTO_SAMPLING_ENABLE macro 33
DRV_ADC_BUFFER_HANDLE type 70
DRV_ADC_BUFFER_STATUS enumeration 71
DRV_ADC_ChannelExtendedScanInputsAdd function 55
DRV_ADC_ChannelExtendedScanInputsRemove function 56
DRV_ADC_ChannelScanInputsAdd function 57
DRV_ADC_ChannelScanInputsRemove function 58
DRV_ADC_CLIENT_STATUS enumeration 64
DRV_ADC_CLIENTS_NUMBER macro 33
DRV_ADC_ClientStatus function 48
DRV_ADC_Close function 49
drv_adc_config_template.h 77
DRV_ADC_CONVERSION_CLOCK_PRESCALER macro 33
DRV_ADC_CONVERSION_CLOCK_SOURCE macro 34
DRV_ADC_CONVERSION_TRIGGER_SOURCE macro 34
DRV_ADC_Deinitialize function 43
DRV_ADC_EVENT enumeration 72
DRV_ADC_EVENT_HANDLER type 73
DRV_ADC_INDEX macro 38
DRV_ADC_INDEX_0 macro 67
DRV_ADC_INDEX_1 macro 68
DRV_ADC_INDEX_2 macro 69
DRV_ADC_INIT structure 65
DRV_ADC_INIT_FLAGS enumeration 66
DRV_ADC_Initialize function 44
DRV_ADC_InputsRegister function 50
DRV_ADC_INSTANCES_NUMBER macro 34
DRV_ADC_INTERNAL_BUFFER_SIZE macro 35
DRV_ADC_INTERRUPT_MODE macro 35
DRV_ADC_INTERRUPT_SOURCE macro 36
DRV_ADC_NegativeInputSelect function 59
DRV_ADC_Open function 51
DRV_ADC_PERIPHERAL_ID macro 36
DRV_ADC_PositiveInputSelect function 60
DRV_ADC_POWER_STATE macro 36
DRV_ADC_RESULT_FORMAT macro 37
DRV_ADC_SAMPLES_PER_INTERRUPT macro 37
DRV_ADC_SamplesAvailable function 52
DRV_ADC_SamplesRead function 61
DRV_ADC_SamplesReadLatest function 62
DRV_ADC_Start function 53
DRV_ADC_Status function 46
DRV_ADC_Stop function 54
DRV_ADC_STOP_ON_CONVERSION_ENABLE macro 37
DRV_ADC_Tasks function 47
DRV_ADC_VOLTAGE_REFERENCE macro 38
drv_ak4384.h 200
DRV_AK4384_AUDIO_DATA_FORMAT enumeration 191
DRV_AK4384_BCLK_BIT_CLK_DIVISOR macro 156
DRV_AK4384_BUFFER_EVENT enumeration 191
DRV_AK4384_BUFFER_EVENT_HANDLER type 192
DRV_AK4384_BUFFER_HANDLE type 193
DRV_AK4384_BUFFER_HANDLE_INVALID macro 197
DRV_AK4384_BufferAddWrite function 181
DRV_AK4384_BufferCombinedQueueSizeGet function 184
DRV_AK4384_BufferEventHandlerSet function 182
DRV_AK4384_BufferProcessedSizeGet function 185
DRV_AK4384_CHANNEL enumeration 193
DRV_AK4384_ChannelOutputInvertDisable function 168
DRV_AK4384_ChannelOutputInvertEnable function 168
DRV_AK4384_CLIENTS_NUMBER macro 150
DRV_AK4384_Close function 166
DRV_AK4384_COMMAND_EVENT_HANDLER type 194
DRV_AK4384_CommandEventHandlerSet function 188
drv_ak4384_config_template.h 203
DRV_AK4384_CONTROL_CLOCK macro 151
DRV_AK4384_COUNT macro 197
DRV_AK4384_DEEMPHASIS_FILTER enumeration 195
DRV_AK4384_DeEmphasisFilterSet function 169
DRV_AK4384_Deinitialize function 162
DRV_AK4384_INDEX_0 macro 198
DRV_AK4384_INDEX_1 macro 198
DRV_AK4384_INDEX_2 macro 198
DRV_AK4384_INDEX_3 macro 199
DRV_AK4384_INDEX_4 macro 199
DRV_AK4384_INDEX_5 macro 199
DRV_AK4384_INIT structure 195
DRV_AK4384_Initialize function 161
DRV_AK4384_INPUT_REFCLOCK macro 152
DRV_AK4384_INSTANCES_NUMBER macro 153
DRV_AK4384_MCLK_MODE enumeration 196
DRV_AK4384_MCLK_SAMPLE_FREQ_MULTPLIER macro 157
DRV_AK4384_MuteOff function 170

DRV_AK4384_MuteOn function 171
DRV_AK4384_Open function 165
DRV_AK4384_SamplingRateGet function 171
DRV_AK4384_SamplingRateSet function 172
DRV_AK4384_SlowRollOffFilterDisable function 173
DRV_AK4384_SlowRollOffFilterEnable function 174
DRV_AK4384_Status function 163
DRV_AK4384_Tasks function 163
DRV_AK4384_TIMER_DRIVER_MODULE_INDEX macro 154
DRV_AK4384_TIMER_PERIOD macro 155
DRV_AK4384_VersionGet function 189
DRV_AK4384_VersionStrGet function 190
DRV_AK4384_VolumeGet function 174
DRV_AK4384_VolumeSet function 175
DRV_AK4384_ZERO_DETECT_MODE enumeration 196
DRV_AK4384_ZeroDetectDisable function 176
DRV_AK4384_ZeroDetectEnable function 177
DRV_AK4384_ZeroDetectInvertDisable function 177
DRV_AK4384_ZeroDetectInvertEnable function 178
DRV_AK4384_ZeroDetectModeSet function 179
drv_ak4642.h 253
DRV_AK4642_AUDIO_DATA_FORMAT enumeration 247
DRV_AK4642_BCLK_BIT_CLK_DIVISOR macro 215
DRV_AK4642_BUFFER_EVENT enumeration 247
DRV_AK4642_BUFFER_EVENT_HANDLER type 248
DRV_AK4642_BUFFER_HANDLE type 249
DRV_AK4642_BUFFER_HANDLE_INVALID macro 245
DRV_AK4642_BufferAddRead function 236
DRV_AK4642_BufferAddWrite function 235
DRV_AK4642_BufferAddWriteRead function 237
DRV_AK4642_BufferEventHandlerSet function 239
DRV_AK4642_CHANNEL enumeration 250
DRV_AK4642_CLIENTS_NUMBER macro 216
DRV_AK4642_Close function 228
DRV_AK4642_COMMAND_EVENT_HANDLER type 250
DRV_AK4642_CommandEventHandlerSet function 242
drv_ak4642_config_template.h 255
DRV_AK4642_COUNT macro 245
DRV_AK4642_Deinitialize function 224
DRV_AK4642_INDEX_0 macro 246
DRV_AK4642_INDEX_1 macro 246
DRV_AK4642_INDEX_2 macro 246
DRV_AK4642_INDEX_3 macro 246
DRV_AK4642_INDEX_4 macro 247
DRV_AK4642_INDEX_5 macro 247
DRV_AK4642_INIT structure 251
DRV_AK4642_Initialize function 223
DRV_AK4642_INPUT_REFCLOCK macro 217
DRV_AK4642_INSTANCES_NUMBER macro 218
DRV_AK4642_INT_EXT_MIC enumeration 252
DRV_AK4642_IntExtMicSet function 233
DRV_AK4642_MCLK_SAMPLE_FREQ_MULTPLIER macro 219
DRV_AK4642_MCLK_SOURCE macro 220
DRV_AK4642_MONO_STEREO_MIC enumeration 252
DRV_AK4642_MonoStereoMicSet function 234
DRV_AK4642_MuteOff function 229
DRV_AK4642_MuteOn function 229
DRV_AK4642_Open function 227
DRV_AK4642_SamplingRateGet function 230
DRV_AK4642_SamplingRateSet function 231
DRV_AK4642_Status function 225
DRV_AK4642_Tasks function 225
DRV_AK4642_VersionGet function 243
DRV_AK4642_VersionStrGet function 244
DRV_AK4642_VolumeGet function 231
DRV_AK4642_VolumeSet function 232
drv_ak4645.h 305
DRV_AK4645_AUDIO_DATA_FORMAT enumeration 297
DRV_AK4645_BCLK_BIT_CLK_DIVISOR macro 267
DRV_AK4645_BUFFER_EVENT enumeration 297
DRV_AK4645_BUFFER_EVENT_HANDLER type 298
DRV_AK4645_BUFFER_HANDLE type 299
DRV_AK4645_BUFFER_HANDLE_INVALID macro 302
DRV_AK4645_BufferAddRead function 287
DRV_AK4645_BufferAddWrite function 288
DRV_AK4645_BufferAddWriteRead function 289
DRV_AK4645_BufferEventHandlerSet function 291
DRV_AK4645_CHANNEL enumeration 299
DRV_AK4645_CLIENTS_NUMBER macro 268
DRV_AK4645_Close function 280
DRV_AK4645_COMMAND_EVENT_HANDLER type 300
DRV_AK4645_CommandEventHandlerSet function 294
drv_ak4645_config_template.h 307
DRV_AK4645_COUNT macro 302
DRV_AK4645_Deinitialize function 276
DRV_AK4645_INDEX_0 macro 303
DRV_AK4645_INDEX_1 macro 303
DRV_AK4645_INDEX_2 macro 303
DRV_AK4645_INDEX_3 macro 303
DRV_AK4645_INDEX_4 macro 304
DRV_AK4645_INDEX_5 macro 304
DRV_AK4645_INIT structure 300
DRV_AK4645_Initialize function 275
DRV_AK4645_INPUT_REFCLOCK macro 269
DRV_AK4645_INSTANCES_NUMBER macro 270
DRV_AK4645_INT_EXT_MIC enumeration 301
DRV_AK4645_IntExtMicSet function 281
DRV_AK4645_MCLK_SAMPLE_FREQ_MULTPLIER macro 271
DRV_AK4645_MCLK_SOURCE macro 272
DRV_AK4645_MONO_STEREO_MIC enumeration 301
DRV_AK4645_MonoStereoMicSet function 281
DRV_AK4645_MuteOff function 282
DRV_AK4645_MuteOn function 283
DRV_AK4645_Open function 279
DRV_AK4645_SamplingRateGet function 283
DRV_AK4645_SamplingRateSet function 284
DRV_AK4645_Status function 277
DRV_AK4645_Tasks function 277
DRV_AK4645_VersionGet function 295
DRV_AK4645_VersionStrGet function 296
DRV_AK4645_VolumeGet function 285
DRV_AK4645_VolumeSet function 285
drv_ak4953.h 356
DRV_AK4953_AUDIO_DATA_FORMAT enumeration 347
DRV_AK4953_BCLK_BIT_CLK_DIVISOR macro 315
DRV_AK4953_BUFFER_EVENT enumeration 347

DRV_AK4953_BUFFER_EVENT_HANDLER type 348
DRV_AK4953_BUFFER_HANDLE type 349
DRV_AK4953_BUFFER_HANDLE_INVALID macro 352
DRV_AK4953_BufferAddRead function 344
DRV_AK4953_BufferAddWrite function 338
DRV_AK4953_BufferAddWriteRead function 339
DRV_AK4953_BufferEventHandlerSet function 331
DRV_AK4953_CHANNEL enumeration 354
DRV_AK4953_CLIENTS_NUMBER macro 316
DRV_AK4953_Close function 328
DRV_AK4953_COMMAND_EVENT_HANDLER type 349
DRV_AK4953_CommandEventHandlerSet function 329
drv_ak4953_config_template.h 359
DRV_AK4953_COUNT macro 352
DRV_AK4953_Deinitialize function 326
DRV_AK4953_DIGITAL_BLOCK_CONTROL enumeration 350
DRV_AK4953_INDEX_0 macro 352
DRV_AK4953_INDEX_1 macro 353
DRV_AK4953_INDEX_2 macro 353
DRV_AK4953_INDEX_3 macro 353
DRV_AK4953_INDEX_4 macro 353
DRV_AK4953_INDEX_5 macro 354
DRV_AK4953_INIT structure 351
DRV_AK4953_Initialize function 325
DRV_AK4953_INPUT_REFCLOCK macro 317
DRV_AK4953_INSTANCES_NUMBER macro 318
DRV_AK4953_INT_EXT_MIC enumeration 354
DRV_AK4953_IntExtMicSet function 345
DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER macro 319
DRV_AK4953_MCLK_SOURCE macro 320
DRV_AK4953_MONO_STEREO_MIC enumeration 355
DRV_AK4953_MonoStereoMicSet function 346
DRV_AK4953_MuteOff function 341
DRV_AK4953_MuteOn function 342
DRV_AK4953_Open function 327
DRV_AK4953_QUEUE_DEPTH_COMBINED macro 321
DRV_AK4953_SamplingRateGet function 334
DRV_AK4953_SamplingRateSet function 333
DRV_AK4953_Status function 334
DRV_AK4953_Tasks function 329
DRV_AK4953_VersionGet function 335
DRV_AK4953_VersionStrGet function 336
DRV_AK4953_VolumeGet function 336
DRV_AK4953_VolumeSet function 343
drv_camera.h 94
DRV_CAMERA_Close function 81
DRV_CAMERA_Deinitialize function 82
DRV_CAMERA_INDEX_0 macro 90
DRV_CAMERA_INDEX_1 macro 91
DRV_CAMERA_INDEX_COUNT macro 92
DRV_CAMERA_INIT structure 88
DRV_CAMERA_Initialize function 83
DRV_CAMERA_INTERRUPT_PORT_REMAP structure 89
DRV_CAMERA_Open function 84
drv_camera_ovm7690.h 124
DRV_CAMERA_OVM7690_CLIENT_OBJ structure 117
DRV_CAMERA_OVM7690_CLIENT_STATUS enumeration 117
DRV_CAMERA_OVM7690_Close function 108
DRV_CAMERA_OVM7690_Deinitialize function 104
DRV_CAMERA_OVM7690_ERROR enumeration 118
DRV_CAMERA_OVM7690_FrameBufferAddressSet function 109
DRV_CAMERA_OVM7690_FrameRectSet function 112
DRV_CAMERA_OVM7690_HsyncEventHandler function 114
DRV_CAMERA_OVM7690_INDEX_0 macro 122
DRV_CAMERA_OVM7690_INDEX_1 macro 122
DRV_CAMERA_OVM7690_INIT structure 118
DRV_CAMERA_OVM7690_Initialize function 103
DRV_CAMERA_OVM7690_OBJ structure 119
DRV_CAMERA_OVM7690_Open function 107
DRV_CAMERA_OVM7690_RECT structure 121
DRV_CAMERA_OVM7690_REG12_OP_FORMAT enumeration 121
DRV_CAMERA_OVM7690_REG12_SOFT_RESET macro 122
DRV_CAMERA_OVM7690_RegisterSet function 105
DRV_CAMERA_OVM7690_SCCB_READ_ID macro 123
DRV_CAMERA_OVM7690_SCCB_WRITE_ID macro 123
DRV_CAMERA_OVM7690_Start function 110
DRV_CAMERA_OVM7690_Stop function 111
DRV_CAMERA_OVM7690_VsyncEventHandler function 115
DRV_CAMERA_Reinitialize function 85
DRV_CAMERA_Status function 86
DRV_CAMERA_Tasks function 87
DRV_CAN_ChannelMessageReceive function 130
DRV_CAN_ChannelMessageTransmit function 131
DRV_CAN_Close function 132
DRV_CAN_Deinitialize function 133
DRV_CAN_Initialize function 134
DRV_CAN_Open function 135
DRV_CLIENT_STATUS enumeration 7
DRV_CMP_Initialize function 363
DRV_CONFIG_NOT_SUPPORTED macro 11
DRV_DYNAMIC_BUILD macro 667
DRV_EBI_Initialize function 386
drv_encx24j600.h 423
DRV_ENCX24J600_Close function 405
DRV_ENCX24J600_ConfigGet function 406
DRV_ENCX24J600_Configuration structure 420
DRV_ENCX24J600_Deinitialize function 398
DRV_ENCX24J600_EventAcknowledge function 417
DRV_ENCX24J600_EventMaskSet function 418
DRV_ENCX24J600_EventPendingGet function 419
DRV_ENCX24J600_Initialize function 399
DRV_ENCX24J600_LinkCheck function 407
DRV_ENCX24J600_MDIX_TYPE enumeration 422
DRV_ENCX24J600_Open function 408
DRV_ENCX24J600_PacketRx function 414
DRV_ENCX24J600_PacketTx function 416
DRV_ENCX24J600_ParametersGet function 409
DRV_ENCX24J600_PowerMode function 410
DRV_ENCX24J600_Process function 404
DRV_ENCX24J600_RegisterStatisticsGet function 411
DRV_ENCX24J600_Reinitialize function 400
DRV_ENCX24J600_RxFilterHashTableEntrySet function 415
DRV_ENCX24J600_SetMacCtrlInfo function 402
DRV_ENCX24J600_StackInitialize function 403
DRV_ENCX24J600_StatisticsGet function 412
DRV_ENCX24J600_Status function 413

DRV_ENCX24J600_Tasks function 401
drv_ethmac.h 460
DRV_ETHMAC_CLIENTS_NUMBER macro 430
drv_ethmac_config.h 461
DRV_ETHMAC_INDEX macro 430
DRV_ETHMAC_INDEX_0 macro 458
DRV_ETHMAC_INDEX_1 macro 457
DRV_ETHMAC_INDEX_COUNT macro 459
DRV_ETHMAC_INSTANCES_NUMBER macro 431
DRV_ETHMAC_INTERRUPT_MODE macro 431
DRV_ETHMAC_INTERRUPT_SOURCE macro 432
DRV_ETHMAC_PERIPHERAL_ID macro 432
DRV_ETHMAC_PIC32MACClose function 436
DRV_ETHMAC_PIC32MACConfigGet function 446
DRV_ETHMAC_PIC32MACDeinitialize function 437
DRV_ETHMAC_PIC32MACEventAcknowledge function 452
DRV_ETHMAC_PIC32MACEventMaskSet function 453
DRV_ETHMAC_PIC32MACEventPendingGet function 454
DRV_ETHMAC_PIC32MACInitialize function 438
DRV_ETHMAC_PIC32MACLinkCheck function 439
DRV_ETHMAC_PIC32MACOpen function 440
DRV_ETHMAC_PIC32MACPacketRx function 449
DRV_ETHMAC_PIC32MACPacketTx function 451
DRV_ETHMAC_PIC32MACParametersGet function 441
DRV_ETHMAC_PIC32MACPowerMode function 442
DRV_ETHMAC_PIC32MACProcess function 443
DRV_ETHMAC_PIC32MACRegisterStatisticsGet function 447
DRV_ETHMAC_PIC32MACReinitialize function 448
DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet function 450
DRV_ETHMAC_PIC32MACStatisticsGet function 444
DRV_ETHMAC_PIC32MACStatus function 445
DRV_ETHMAC_PIC32MACTasks function 456
DRV_ETHMAC_POWER_STATE macro 432
DRV_ETHMAC_Tasks_ISR function 455
drv_ethphy.h 519
DRV_ETHPHY_CLIENT_STATUS enumeration 504
DRV_ETHPHY_ClientOperationAbort function 489
DRV_ETHPHY_ClientOperationResult function 490
DRV_ETHPHY_CLIENTS_NUMBER macro 467
DRV_ETHPHY_ClientStatus function 483
DRV_ETHPHY_Close function 484
drv_ethphy_config.h 521
DRV_ETHPHY_CONFIG_FLAGS enumeration 516
DRV_ETHPHY_Deinitialize function 475
DRV_ETHPHY_HWConfigFlagsGet function 481
DRV_ETHPHY_INDEX macro 467
DRV_ETHPHY_INDEX_0 macro 511
DRV_ETHPHY_INDEX_1 macro 512
DRV_ETHPHY_INDEX_COUNT macro 513
DRV_ETHPHY_INIT structure 505
DRV_ETHPHY_Initialize function 474
DRV_ETHPHY_INSTANCES_NUMBER macro 468
DRV_ETHPHY_LINK_STATUS enumeration 515
DRV_ETHPHY_LinkStatusGet function 514
DRV_ETHPHY_NEG_DONE_TMO macro 468
DRV_ETHPHY_NEG_INIT_TMO macro 469
DRV_ETHPHY_NEGOTIATION_RESULT structure 506
DRV_ETHPHY_NegotiationIsComplete function 485
DRV_ETHPHY_NegotiationResultGet function 476
DRV_ETHPHY_OBJECT structure 517
DRV_ETHPHY_Open function 486
DRV_ETHPHY_PERIPHERAL_ID macro 468
DRV_ETHPHY_PhyAddressGet function 477
DRV_ETHPHY_Reinitialize function 478
DRV_ETHPHY_Reset function 487
DRV_ETHPHY_RESET_CLR_TMO macro 469
DRV_ETHPHY_RestartNegotiation function 488
DRV_ETHPHY_Setup function 482
DRV_ETHPHY_SETUP structure 507
DRV_ETHPHY_SMIClockSet function 493
DRV_ETHPHY_SMIRead function 495
DRV_ETHPHY_SMIScanDataGet function 496
DRV_ETHPHY_SMIScanStart function 494
DRV_ETHPHY_SMIScanStatusGet function 491
DRV_ETHPHY_SMIScanStop function 492
DRV_ETHPHY_SMIStatus function 497
DRV_ETHPHY_SMIWrite function 498
DRV_ETHPHY_Status function 479
DRV_ETHPHY_Tasks function 480
DRV_ETHPHY_VENDOR_MDIX_CONFIGURE type 508
DRV_ETHPHY_VENDOR_MII_CONFIGURE type 509
DRV_ETHPHY_VENDOR_SMI_CLOCK_GET type 510
DRV_ETHPHY_VENDOR_WOL_CONFIGURE type 518
DRV_ETHPHY_VendorDataGet function 499
DRV_ETHPHY_VendorDataSet function 500
DRV_ETHPHY_VendorSMIReadResultGet function 501
DRV_ETHPHY_VendorSMIReadStart function 502
DRV_ETHPHY_VendorSMIWriteStart function 503
DRV_GFX_CONFIG_LCC_EXTERNAL_MEMORY macro 529
DRV_GFX_CONFIG_LCC_INTERNAL_MEMORY macro 530
DRV_GFX_CONFIG_LCC_PALETTE macro 531
DRV_GFX_INIT structure 543
drv_gfx_lcc.h 649
DRV_GFX_LCC_AlphaBlendWindow function 549
DRV_GFX_LCC_BarFill function 551
DRV_GFX_LCC_Close function 552
drv_gfx_lcc_config_template.h 655
DRV_GFX_LCC_DISPLAY_WRITE_BUS_TYPE enumeration 567
DRV_GFX_LCC_DisplayRefresh function 550
DRV_GFX_LCC_DMA_CHANNEL_INDEX macro 532
DRV_GFX_LCC_FB_WRITE_BUS_TYPE enumeration 568
DRV_GFX_LCC_FrameBufferAddressSet function 563
DRV_GFX_LCC_GetBuffer function 553
DRV_GFX_LCC_INDEX_COUNT macro 566
DRV_GFX_LCC_Initialize function 554
DRV_GFX_LCC_InterfaceSet function 555
DRV_GFX_LCC_MaxXGet function 561
DRV_GFX_LCC_MaxYGet function 562
DRV_GFX_LCC_Open function 556
DRV_GFX_LCC_PixelArrayGet function 557
DRV_GFX_LCC_PixelArrayPut function 558
DRV_GFX_LCC_PixelPut function 546
DRV_GFX_LCC_SetColor function 547
DRV_GFX_LCC_SetPage function 548
DRV_GFX_LCC_Tasks function 559
DRV_GFX_LCD enumeration 544

drv_gfx_otm2201a.h 652
DRV_GFX_OTM2201A_AddressSet function 608
DRV_GFX_OTM2201A_BarFill function 617
DRV_GFX_OTM2201A_BrightnessSet function 609
DRV_GFX_OTM2201A_Busy function 610
DRV_GFX_OTM2201A_Close function 618
DRV_GFX_OTM2201A_ColorSet function 611
DRV_GFX_OTM2201A_COMMAND structure 627
drv_gfx_otm2201a_config_template.h 656
DRV_GFX_OTM2201A_INDEX_COUNT macro 628
DRV_GFX_OTM2201A_Initialize function 619
DRV_GFX_OTM2201A_InstanceSet function 612
DRV_GFX_OTM2201A_InterfaceSet function 620
DRV_GFX_OTM2201A_MaxXGet function 624
DRV_GFX_OTM2201A_MaxYGet function 625
DRV_GFX_OTM2201A_Open function 621
DRV_GFX_OTM2201A_PixelArrayGet function 613
DRV_GFX_OTM2201A_PixelArrayPut function 614
DRV_GFX_OTM2201A_PixelPut function 615
DRV_GFX_OTM2201A_RegGet function 616
DRV_GFX_OTM2201A_RegSet function 622
DRV_GFX_OTM2201A_Tasks function 623
DRV_GFX_PaletteSet function 560
drv_gfx_s1d13517.h 650
DRV_GFX_S1D13517_AlphaBlendWindow function 569
DRV_GFX_S1D13517_BarFill function 579
DRV_GFX_S1D13517_BrightnessSet function 570
DRV_GFX_S1D13517_Close function 580
drv_gfx_s1d13517_config_template.h 655
DRV_GFX_S1D13517_GetReg function 571
DRV_GFX_S1D13517_INDEX_COUNT macro 588
DRV_GFX_S1D13517_Initialize function 581
DRV_GFX_S1D13517_InterfaceSet function 582
DRV_GFX_S1D13517_Layer function 572
DRV_GFX_S1D13517_MaxXGet function 585
DRV_GFX_S1D13517_MaxYGet function 586
DRV_GFX_S1D13517_Open function 583
DRV_GFX_S1D13517_PixelArrayPut function 573
DRV_GFX_S1D13517_PixelPut function 574
DRV_GFX_S1D13517_SetColor function 575
DRV_GFX_S1D13517_SetInstance function 576
DRV_GFX_S1D13517_SetPage function 577
DRV_GFX_S1D13517_SetReg function 578
DRV_GFX_S1D13517_Tasks function 584
drv_gfx_ssd1289.h 651
drv_gfx_ssd1926.h 652
DRV_GFX_SSD1926_BarFill function 590
DRV_GFX_SSD1926_Busy function 591
DRV_GFX_SSD1926_Close function 599
DRV_GFX_SSD1926_COMMAND structure 1675
drv_gfx_ssd1926_config_template.h 655
DRV_GFX_SSD1926_GetReg function 592
DRV_GFX_SSD1926_INDEX_COUNT macro 607
DRV_GFX_SSD1926_Initialize function 600
DRV_GFX_SSD1926_InterfaceSet function 601
DRV_GFX_SSD1926_MaxXGet function 605
DRV_GFX_SSD1926_MaxYGet function 606
DRV_GFX_SSD1926_Open function 602
DRV_GFX_SSD1926_PixelArrayGet function 593
DRV_GFX_SSD1926_PixelArrayPut function 594
DRV_GFX_SSD1926_PixelPut function 595
DRV_GFX_SSD1926_SetColor function 596
DRV_GFX_SSD1926_SetInstance function 597
DRV_GFX_SSD1926_SetReg function 598
DRV_GFX_SSD1926_Status function 603
DRV_GFX_SSD1926_Tasks function 604
drv_gfx_tft002.h 654
DRV_GFX_TFT002_BrightnessSet function 629
DRV_GFX_TFT002_Busy function 630
DRV_GFX_TFT002_Close function 631
DRV_GFX_TFT002_COMMAND structure 647
drv_gfx_tft002_config_template.h 656
DRV_GFX_TFT002_GetReg function 632
DRV_GFX_TFT002_INDEX_COUNT macro 648
DRV_GFX_TFT002_Initialize function 633
DRV_GFX_TFT002_InterfaceGet function 634
DRV_GFX_TFT002_MaxXGet function 635
DRV_GFX_TFT002_MaxYGet function 636
DRV_GFX_TFT002_Open function 637
DRV_GFX_TFT002_PixelArrayGet function 638
DRV_GFX_TFT002_PixelArrayPut function 639
DRV_GFX_TFT002_PixelPut function 640
DRV_GFX_TFT002_PixelsPut function 641
DRV_GFX_TFT002_SetColor function 642
DRV_GFX_TFT002_SetInstance function 643
DRV_GFX_TFT002_SetReg function 644
DRV_GFX_TFT002_Status function 645
DRV_GFX_TFT002_Tasks function 646
DRV_HANDLE type 8
DRV_HANDLE_INVALID macro 12
drv_i2c.h 738
DRV_I2C_ADDRESS_WIDTH enumeration 712
DRV_I2C_BaudRateSet function 682
DRV_I2C_BUFFER_EVENT enumeration 713
DRV_I2C_BUFFER_EVENT_HANDLER type 714
DRV_I2C_BUFFER_HANDLE type 715
DRV_I2C_BUFFER_QUEUE_SUPPORT macro 734
DRV_I2C_BufferAddRead function 698
DRV_I2C_BufferAddWrite function 699
DRV_I2C_BufferAddWriteRead function 700
DRV_I2C_BufferEventHandlerSet function 693
DRV_I2C_BufferStatus function 692
DRV_I2C_BUS_LEVEL enumeration 716
DRV_I2C_BUS_SPEED enumeration 717
DRV_I2C_ByteRead function 695
DRV_I2C_ByteWrite function 696
DRV_I2C_CallBack type 718
DRV_I2C_Close function 689
DRV_I2C_CONFIG_BUILD_TYPE macro 668
DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BASIC macro 668
DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BLOCKING macro 669
DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_EXCLUSIVE macro 669
DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_MASTER macro 669

© 2013-2015 Microchip Technology Inc.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_NON_BLOCKING macro 670
DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_READ macro 670
DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_SLAVE macro 671
DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE macro 671
DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE_READ macro 671
drv_i2c_config_template.h 740
DRV_I2C_Deinitialize function 677
DRV_I2C_INDEX macro 354
DRV_I2C_INDEX_0 macro 728
DRV_I2C_INDEX_1 macro 729
DRV_I2C_INDEX_2 macro 730
DRV_I2C_INDEX_3 macro 731
DRV_I2C_INDEX_4 macro 732
DRV_I2C_INDEX_5 macro 733
DRV_I2C_INIT structure 719
DRV_I2C_INIT_CODE enumeration 721
DRV_I2C_Initialize function 678
DRV_I2C_INSTANCES_NUMBER macro 735
DRV_I2C_INTERRUPT_MODE macro 736
DRV_I2C_IRQEventSend function 711
DRV_I2C_MasterACKSend function 683
DRV_I2C_MasterBusIdle function 684
DRV_I2C_MasterNACKSend function 685
DRV_I2C_MasterRestart function 688
DRV_I2C_MasterStart function 686
DRV_I2C_MasterStop function 687
DRV_I2C_MODE enumeration 722
DRV_I2C_Open function 690
DRV_I2C_QUEUE_DEPTH_COMBINED macro 737
DRV_I2C_ReceiverBufferIsEmpty function 709
DRV_I2C_RestartEventSend function 702
DRV_I2C_SetUpByteRead function 697
DRV_I2C_SLAVE_ADDRESS_MASK type 723
DRV_I2C_Status function 680
DRV_I2C_StopEventSend function 703
DRV_I2C_Tasks function 681
DRV_I2C_TRANSFER_HANDLE type 724
DRV_I2C_WaitForACKOrNACKComplete function 710
DRV_I2C_WaitForByteWriteToComplete function 704
DRV_I2C_WaitForReadByteAvailable function 705
DRV_I2C_WaitForStartComplete function 706
DRV_I2C_WaitForStopComplete function 707
DRV_I2C_WriteByteAcknowledged function 708
drv_i2s.h 833
DRV_I2S_AUDIO_PROTOCOL_MODE enumeration 810
DRV_I2S_BaudSet function 802
DRV_I2S_BUFFER_EVENT enumeration 811
DRV_I2S_BUFFER_EVENT_HANDLER type 812
DRV_I2S_BUFFER_HANDLE type 814
DRV_I2S_BUFFER_HANDLE_INVALID macro 823
DRV_I2S_BufferAddRead function 786
DRV_I2S_BufferAddWrite function 788
DRV_I2S_BufferAddWriteRead function 790
DRV_I2S_BufferCombinedQueueSizeGet function 796
DRV_I2S_BufferEventHandlerSet function 792
DRV_I2S_BufferProcessedSizeGet function 794
DRV_I2S_CLIENTS_NUMBER macro 772
DRV_I2S_CLOCK_MODE enumeration 815
DRV_I2S_Close function 783
drv_i2s_config_template.h 835
DRV_I2S_COUNT macro 824
DRV_I2S_DATA16 structure 816
DRV_I2S_DATA24 structure 817
DRV_I2S_DATA32 structure 818
DRV_I2S_Deinitialize function 777
DRV_I2S_ERROR enumeration 819
DRV_I2S_ErrorGet function 804
DRV_I2S_INDEX macro 761
DRV_I2S_INDEX_0 macro 827
DRV_I2S_INDEX_1 macro 828
DRV_I2S_INDEX_2 macro 829
DRV_I2S_INDEX_3 macro 830
DRV_I2S_INDEX_4 macro 831
DRV_I2S_INDEX_5 macro 832
DRV_I2S_INIT structure 820
DRV_I2S_Initialize function 778
DRV_I2S_INSTANCES_NUMBER macro 762
DRV_I2S_INTERRUPT_MODE macro 763
DRV_I2S_INTERRUPT_SOURCE_ERROR macro 764
DRV_I2S_INTERRUPT_SOURCE_RECEIVE macro 765
DRV_I2S_INTERRUPT_SOURCE_TRANSMIT macro 766
DRV_I2S_MODE enumeration 822
DRV_I2S_Open function 784
DRV_I2S_PERIPHERAL_ID macro 767
DRV_I2S_QUEUE_DEPTH_COMBINED macro 773
DRV_I2S_Read function 798
DRV_I2S_READ_ERROR macro 825
DRV_I2S_RECEIVE_DMA_CHAINING_CHANNEL macro 771
DRV_I2S_RECEIVE_DMA_CHANNEL macro 768
DRV_I2S_ReceiveErrorIgnore function 806
DRV_I2S_Status function 780
DRV_I2S_STOP_IN_IDLE macro 769
DRV_I2S_Tasks function 781
DRV_I2S_TasksError function 782
DRV_I2S_TRANSMIT_DMA_CHANNEL macro 770
DRV_I2S_TransmitErrorIgnore function 808
DRV_I2S_Write function 800
DRV_I2S_WRITE_ERROR macro 826
DRV_IC_BufferIsEmpty function 841
DRV_IC_Capture16BitDataRead function 842
DRV_IC_Capture32BitDataRead function 843
DRV_IC_Initialize function 840
DRV_IC_Start function 844
DRV_IC_Stop function 845
DRV_IO_BUFFER_TYPES enumeration 9
DRV_IO_INTENT enumeration 10
DRV_IO_ISBLOCKING macro 13
DRV_IO_ISEXCLUSIVE macro 14
DRV_IO_ISNONBLOCKING macro 15
drv_mtch6301.h 885
DRV_MTCH6301_CALIBRATION_DELAY macro 854
DRV_MTCH6301_CALIBRATION_INSET macro 854

DRV_MTCH6301_CLIENTS_NUMBER macro 855
drv_mtch6301_config_template.h 886
DRV_MTCH6301_INDEX macro 855
DRV_MTCH6301_INSTANCES_NUMBER macro 855
DRV_MTCH6301_INTERRUPT_MODE macro 856
DRV_MTCH6301_SAMPLE_POINTS macro 856
DRV_MTCH6301_TOUCH_DIAMETER macro 856
drv_nvm.h 943
DRV_NVM_AddressGet function 929
DRV_NVM_BUFFER_OBJECT_NUMBER macro 903
DRV_NVM_CLIENTS_NUMBER macro 904
DRV_NVM_Close function 917
DRV_NVM_COMMAND_HANDLE type 940
DRV_NVM_COMMAND_HANDLE_INVALID macro 942
DRV_NVM_COMMAND_STATUS enumeration 941
DRV_NVM_CommandStatus function 930
drv_nvm_config_template.h 944
DRV_NVM_Deinitialize function 914
DRV_NVM_DISABLE_ERROR_CHECK macro 907
DRV_NVM_Erase function 922
DRV_NVM_ERASE_WRITE_ENABLE macro 906
DRV_NVM_EraseWrite function 924
DRV_NVM_EVENT enumeration 937
DRV_NVM_EVENT_HANDLER type 938
DRV_NVM_EventHandlerSet function 926
DRV_NVM_GeometryGet function 931
DRV_NVM_INDEX_0 macro 934
DRV_NVM_INDEX_1 macro 936
DRV_NVM_INIT structure 935
DRV_NVM_Initialize function 912
DRV_NVM_INSTANCES_NUMBER macro 904
DRV_NVM_INTERRUPT_MODE macro 904
DRV_NVM_IsAttached function 932
DRV_NVM_IsWriteProtected function 933
DRV_NVM_MEDIA_SIZE macro 907
DRV_NVM_MEDIA_START_ADDRESS macro 907
DRV_NVM_Open function 916
DRV_NVM_PAGE_SIZE macro 906
DRV_NVM_Read function 918
DRV_NVM_ROW_SIZE macro 905
DRV_NVM_Status function 915
DRV_NVM_SYS_FS_REGISTER macro 908
DRV_NVM_Tasks function 928
DRV_NVM_UNLOCK_KEY1 macro 905
DRV_NVM_UNLOCK_KEY2 macro 905
DRV_NVM_Write function 920
DRV_OC_Disable function 949
DRV_OC_Enable function 950
DRV_OC_FaultHasOccurred function 951
DRV_OC_Initialize function 952
DRV_OC_Start function 953
DRV_OC_Stop function 954
drv_ovm7690_config_template.h 126
DRV_OVM7690_INTERRUPT_MODE macro 100
drv_pmp.h 1002
DRV_PMP_CHIPX_STROBE_MODE enumeration 987
DRV_PMP_CLIENT_STATUS enumeration 988
DRV_PMP_CLIENTS_NUMBER macro 965
DRV_PMP_ClientStatus function 978
DRV_PMP_Close function 979
drv_pmp_config.h 1003
DRV_PMP_Deinitialize function 970
DRV_PMP_ENDIAN_MODE enumeration 989
DRV_PMP_INDEX enumeration 990
DRV_PMP_INDEX_COUNT macro 986
DRV_PMP_INIT structure 991
DRV_PMP_Initialize function 971
DRV_PMP_INSTANCES_NUMBER macro 965
DRV_PMP_MODE_CONFIG structure 992
DRV_PMP_ModeConfig function 980
DRV_PMP_Open function 981
DRV_PMP_POLARITY_OBJECT structure 993
DRV_PMP_PORT_CONTROL enumeration 994
DRV_PMP_PORTS structure 995
DRV_PMP_QUEUE_ELEMENT_OBJ structure 996
DRV_PMP_QUEUE_SIZE macro 966
DRV_PMP_Read function 982
DRV_PMP_Reinitialize function 973
DRV_PMP_Status function 975
DRV_PMP_Tasks function 976
DRV_PMP_TimingSet function 977
DRV_PMP_TRANSFER_STATUS enumeration 997
DRV_PMP_TRANSFER_TYPE enumeration 1000
DRV_PMP_TransferStatus function 985
DRV_PMP_WAIT_STATES structure 998
DRV_PMP_Write function 983
DRV_RTCC_AlarmDateGet function 1008
DRV_RTCC_AlarmTimeGet function 1009
DRV_RTCC_ClockOutput function 1010
DRV_RTCC_DateGet function 1011
DRV_RTCC_Initialize function 1012
DRV_RTCC_Start function 1013
DRV_RTCC_Stop function 1014
DRV_RTCC_TimeGet function 1015
drv_sdcard.h 1061
DRV_SDCARD_CLIENTS_NUMBER macro 1024
DRV_SDCARD_Close function 1035
DRV_SDCARD_COMMAND_HANDLE type 1056
DRV_SDCARD_COMMAND_HANDLE_INVALID macro 1055
DRV_SDCARD_COMMAND_STATUS enumeration 1057
DRV_SDCARD_CommandStatus function 1045
drv_sdcard_config_template.h 1062
DRV_SDCARD_Deinitialize function 1030
DRV_SDCARD_EVENT enumeration 1058
DRV_SDCARD_EVENT_HANDLER type 1059
DRV_SDCARD_EventHandlerSet function 1041
DRV_SDCARD_GeometryGet function 1046
DRV_SDCARD_INDEX_0 macro 1047
DRV_SDCARD_INDEX_1 macro 1052
DRV_SDCARD_INDEX_2 macro 1053
DRV_SDCARD_INDEX_3 macro 1054
DRV_SDCARD_INDEX_COUNT macro 1048
DRV_SDCARD_INDEX_MAX macro 1024
DRV_SDCARD_INIT structure 1049
DRV_SDCARD_Initialize function 1029
DRV_SDCARD_INSTANCES_NUMBER macro 1025

DRV_SDCARD_IsAttached function 1043
DRV_SDCARD_IsWriteProtected function 1044
DRV_SDCARD_Open function 1036
DRV_SDCARD_POWER_STATE macro 1025
DRV_SDCARD_Read function 1037
DRV_SDCARD_Reinitialize function 1031
DRV_SDCARD_Status function 1032
DRV_SDCARD_Tasks function 1034
DRV_SDCARD_Write function 1039
drv_spi.h 1136
DRV_SPI_16BIT macro 1074
DRV_SPI_32BIT macro 1075
DRV_SPI_8BIT macro 1076
DRV_SPI_BUFFER_EVENT enumeration 1119
DRV_SPI_BUFFER_EVENT_HANDLER type 1120
DRV_SPI_BUFFER_HANDLE type 1122
DRV_SPI_BUFFER_HANDLE_INVALID macro 1116
DRV_SPI_BUFFER_TYPE enumeration 1123
DRV_SPI_BufferAddRead function 1102
DRV_SPI_BufferAddRead2 function 1108
DRV_SPI_BufferAddWrite function 1104
DRV_SPI_BufferAddWrite2 function 1110
DRV_SPI_BufferAddWriteRead function 1106
DRV_SPI_BufferAddWriteRead2 function 1112
DRV_SPI_BufferStatus function 1101
DRV_SPI_CLIENT_DATA structure 1135
DRV_SPI_ClientConfigure function 1100
DRV_SPI_CLIENTS_NUMBER macro 1088
DRV_SPI_CLOCK_MODE enumeration 1124
DRV_SPI_Close function 1097
drv_spi_config_template.h 1138
DRV_SPI_Deinitialize function 1094
DRV_SPI_DMA macro 1077
DRV_SPI_DMA_DUMMY_BUFFER_SIZE macro 1078
DRV_SPI_DMA_TXFER_SIZE macro 1079
DRV_SPI_EBM macro 1080
DRV_SPI_ELEMENTS_PER_QUEUE macro 1081
DRV_SPI_INDEX_0 macro 1117
DRV_SPI_INDEX_1 macro 1129
DRV_SPI_INDEX_2 macro 1130
DRV_SPI_INDEX_3 macro 1131
DRV_SPI_INDEX_4 macro 1132
DRV_SPI_INDEX_5 macro 1133
DRV_SPI_INDEX_COUNT macro 1118
DRV_SPI_INIT structure 1125
DRV_SPI_Initialize function 1092
DRV_SPI_INSTANCES_NUMBER macro 1087
DRV_SPI_ISR macro 1082
DRV_SPI_MASTER macro 1083
DRV_SPI_MODE enumeration 1127
DRV_SPI_Open function 1098
DRV_SPI_POLLED macro 1084
DRV_SPI_PROTOCOL_TYPE enumeration 1128
DRV_SPI_RM macro 1085
DRV_SPI_SLAVE macro 1086
DRV_SPI_Status function 1095
DRV_SPI_TASK_MODE enumeration 1134
DRV_SPI_Tasks function 1096
DRV_SPIn_ReceiverBufferIsFull function 1114
DRV_SPIn_TransmitterBufferIsFull function 1115
drv_sst25vf016b.h 1244
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE type 1187
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID macro 1190
DRV_SST25VF016B_BLOCK_EVENT enumeration 1187
DRV_SST25VF016B_BlockErase function 1177
DRV_SST25VF016B_BlockEventHandlerSet function 1178
DRV_SST25VF016B_BlockRead function 1180
DRV_SST25VF016B_BlockWrite function 1182
DRV_SST25VF016B_CLIENT_STATUS enumeration 1188
DRV_SST25VF016B_CLIENTS_NUMBER macro 1149
DRV_SST25VF016B_ClientStatus function 1175
DRV_SST25VF016B_Close function 1174
drv_sst25vf016b_config_template.h 1245
DRV_SST25VF016B_Deinitialize function 1171
DRV_SST25VF016B_EVENT_HANDLER type 1188
DRV_SST25VF016B_GeometryGet function 1185
DRV_SST25VF016B_HARDWARE_HOLD_ENABLE macro 1150
DRV_SST25VF016B_HARDWARE_WRITE_PROTECTION_ENABLE macro 1151
DRV_SST25VF016B_INDEX_0 macro 1191
DRV_SST25VF016B_INDEX_1 macro 1191
DRV_SST25VF016B_INIT structure 1189
DRV_SST25VF016B_Initialize function 1170
DRV_SST25VF016B_INSTANCES_NUMBER macro 1152
DRV_SST25VF016B_MediaIsAttached function 1186
DRV_SST25VF016B_MODE macro 1153
DRV_SST25VF016B_Open function 1174
DRV_SST25VF016B_QUEUE_DEPTH_COMBINED macro 1154
DRV_SST25VF016B_Status function 1172
DRV_SST25VF016B_Tasks function 1173
drv_sst25vf020b.h 1246
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE type 1213
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID macro 1217
DRV_SST25VF020B_BLOCK_EVENT enumeration 1213
DRV_SST25VF020B_BlockErase function 1202
DRV_SST25VF020B_BlockEraseWrite function 1209
DRV_SST25VF020B_BlockEventHandlerSet function 1203
DRV_SST25VF020B_BlockRead function 1205
DRV_SST25VF020B_BlockWrite function 1207
DRV_SST25VF020B_CLIENT_STATUS enumeration 1214
DRV_SST25VF020B_CLIENTS_NUMBER macro 1155
DRV_SST25VF020B_ClientStatus function 1198
DRV_SST25VF020B_Close function 1199
DRV_SST25VF020B_COMMAND_STATUS enumeration 1216
DRV_SST25VF020B_CommandStatus function 1198
drv_sst25vf020b_config_template.h 1248
DRV_SST25VF020B_Deinitialize function 1195
DRV_SST25VF020B_EVENT_HANDLER type 1214
DRV_SST25VF020B_GeometryGet function 1211
DRV_SST25VF020B_HARDWARE_HOLD_ENABLE macro 1156
DRV_SST25VF020B_HARDWARE_WRITE_PROTECTION_ENABLE macro 1157
DRV_SST25VF020B_INDEX_0 macro 1217
DRV_SST25VF020B_INDEX_1 macro 1218
DRV_SST25VF020B_INIT structure 1215

© 2013-2015 Microchip Technology Inc.

DRV_SST25VF020B_Initialize function 1194
DRV_SST25VF020B_INSTANCES_NUMBER macro 1158
DRV_SST25VF020B_MediaIsAttached function 1212
DRV_SST25VF020B_MODE macro 1159
DRV_SST25VF020B_Open function 1200
DRV_SST25VF020B_QUEUE_DEPTH_COMBINED macro 1160
DRV_SST25VF020B_Status function 1196
DRV_SST25VF020B_Tasks function 1197
drv_sst25vf064c.h 1248
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE type 1238
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID macro 1242
DRV_SST25VF064C_BLOCK_EVENT enumeration 1238
DRV_SST25VF064C_BlockErase function 1228
DRV_SST25VF064C_BlockEventHandlerSet function 1229
DRV_SST25VF064C_BlockRead function 1231
DRV_SST25VF064C_BlockWrite function 1233
DRV_SST25VF064C_CLIENT_STATUS enumeration 1239
DRV_SST25VF064C_CLIENTS_NUMBER macro 1161
DRV_SST25VF064C_ClientStatus function 1224
DRV_SST25VF064C_Close function 1224
DRV_SST25VF064C_COMMAND_STATUS enumeration 1239
DRV_SST25VF064C_CommandStatus function 1225
drv_sst25vf064c_config_template.h 1250
DRV_SST25VF064C_Deinitialize function 1221
DRV_SST25VF064C_EVENT_HANDLER type 1240
DRV_SST25VF064C_GeometryGet function 1236
DRV_SST25VF064C_HARDWARE_HOLD_ENABLE macro 1162
DRV_SST25VF064C_HARDWARE_WRITE_PROTECTION_ENABLE macro 1163
DRV_SST25VF064C_INDEX_0 macro 1242
DRV_SST25VF064C_INDEX_1 macro 1243
DRV_SST25VF064C_INIT structure 1241
DRV_SST25VF064C_Initialize function 1220
DRV_SST25VF064C_INSTANCES_NUMBER macro 1164
DRV_SST25VF064C_MediaIsAttached function 1237
DRV_SST25VF064C_MODE macro 1165
DRV_SST25VF064C_Open function 1226
DRV_SST25VF064C_QUEUE_DEPTH_COMBINED macro 1166
DRV_SST25VF064C_Status function 1222
DRV_SST25VF064C_Tasks function 1222
DRV_STATIC_BUILD macro 672
drv_tmr.h 1339
DRV_TMR_Alarm16BitDeregister function 1285
DRV_TMR_Alarm16BitRegister function 1296
DRV_TMR_Alarm32BitDeregister function 1287
DRV_TMR_Alarm32BitRegister function 1297
DRV_TMR_AlarmDeregister function 1298
DRV_TMR_AlarmDisable function 1294
DRV_TMR_AlarmEnable function 1295
DRV_TMR_AlarmHasElapsed function 1289
DRV_TMR_AlarmPeriod16BitGet function 1290
DRV_TMR_AlarmPeriod16BitSet function 1291
DRV_TMR_AlarmPeriod32BitGet function 1292
DRV_TMR_AlarmPeriod32BitSet function 1293
DRV_TMR_AlarmPeriodGet function 1300
DRV_TMR_AlarmPeriodSet function 1301
DRV_TMR_AlarmRegister function 1302
DRV_TMR_ASYNC_WRITE_ENABLE macro 1267
DRV_TMR_CALLBACK type 1322
DRV_TMR_CLIENT_STATUS enumeration 1324
DRV_TMR_CLIENTS_NUMBER macro 1267
DRV_TMR_ClientStatus function 1280
DRV_TMR_CLK_SOURCES enumeration 1325
DRV_TMR_CLOCK_PRESCALER macro 1265
DRV_TMR_CLOCK_SOURCE macro 1267
DRV_TMR_ClockSet function 1278
DRV_TMR_Close function 1281
drv_tmr_config_template.h 1341
DRV_TMR_CounterClear function 1313
DRV_TMR_CounterFrequencyGet function 1304
DRV_TMR_CounterValue16BitGet function 1305
DRV_TMR_CounterValue16BitSet function 1307
DRV_TMR_CounterValue32BitGet function 1309
DRV_TMR_CounterValue32BitSet function 1311
DRV_TMR_CounterValueGet function 1314
DRV_TMR_CounterValueSet function 1316
DRV_TMR_Deinitialize function 1273
DRV_TMR_DIVIDER_RANGE structure 1326
DRV_TMR_DividerRangeGet function 1320
DRV_TMR_GateModeClear function 1317
DRV_TMR_GateModeSet function 1279
DRV_TMR_INDEX_0 macro 1329
DRV_TMR_INDEX_1 macro 1330
DRV_TMR_INDEX_2 macro 1331
DRV_TMR_INDEX_3 macro 1332
DRV_TMR_INDEX_4 macro 1333
DRV_TMR_INDEX_5 macro 1334
DRV_TMR_INDEX_6 macro 1335
DRV_TMR_INDEX_7 macro 1336
DRV_TMR_INDEX_8 macro 1337
DRV_TMR_INDEX_9 macro 1338
DRV_TMR_INDEX_COUNT macro 1328
DRV_TMR_INIT structure 1323
DRV_TMR_Initialize function 1274
DRV_TMR_INSTANCES_NUMBER macro 1264
DRV_TMR_INTERRUPT_MODE macro 1264
DRV_TMR_INTERRUPT_SOURCE macro 1266
DRV_TMR_MODE macro 1265
DRV_TMR_MODULE_ID macro 1266
DRV_TMR_MODULE_INIT macro 1266
DRV_TMR_Open function 1282
DRV_TMR_OPERATION_MODE enumeration 1327
DRV_TMR_OperationModeGet function 1319
DRV_TMR_PrescalerGet function 1318
DRV_TMR_Start function 1283
DRV_TMR_Status function 1276
DRV_TMR_Stop function 1284
DRV_TMR_Tasks function 1277
DRV_TMR_Tasks_ISR function 1321
DRV_TOUCH_MTCH6301_CLIENT_OBJECT structure 872
DRV_TOUCH_MTCH6301_Close function 861
DRV_TOUCH_MTCH6301_Deinitialize function 862
DRV_TOUCH_MTCH6301_HANDLE type 873
DRV_TOUCH_MTCH6301_HANDLE_INVALID macro 878
DRV_TOUCH_MTCH6301_I2C_MASTER_READ_ID macro 879

© 2013-2015 Microchip Technology Inc.

DRV_TOUCH_MTCH6301_I2C_MASTER_WRITE_ID macro 880
DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE macro 881
DRV_TOUCH_MTCH6301_INDEX_0 macro 882
DRV_TOUCH_MTCH6301_INDEX_1 macro 883
DRV_TOUCH_MTCH6301_INDEX_COUNT macro 884
DRV_TOUCH_MTCH6301_Initialize function 863
DRV_TOUCH_MTCH6301_MODULE_ID enumeration 874
DRV_TOUCH_MTCH6301_OBJECT structure 875
DRV_TOUCH_MTCH6301_Open function 865
DRV_TOUCH_MTCH6301_ReadRequest function 869
DRV_TOUCH_MTCH6301_Status function 867
DRV_TOUCH_MTCH6301_TASK_QUEUE structure 876
DRV_TOUCH_MTCH6301_TASK_STATE enumeration 877
DRV_TOUCH_MTCH6301_Tasks function 868
DRV_TOUCH_MTCH6301_TouchGetX function 870
DRV_TOUCH_MTCH6301_TouchGetY function 871
drv_usart.h 1432
DRV_USART_BAUD_RATE macro 1362
DRV_USART_BAUD_SET_RESULT enumeration 1418
DRV_USART_BaudSet function 1387
DRV_USART_BUFFER_EVENT enumeration 1419
DRV_USART_BUFFER_EVENT_HANDLER type 1420
DRV_USART_BUFFER_HANDLE type 1422
DRV_USART_BUFFER_HANDLE_INVALID macro 1426
DRV_USART_BUFFER_QUEUE_SUPPORT macro 1365
DRV_USART_BufferAddRead function 1389
DRV_USART_BufferAddWrite function 1391
DRV_USART_BufferEventHandlerSet function 1393
DRV_USART_BufferProcessedSizeGet function 1395
DRV_USART_BYTE_MODEL_SUPPORT macro 1365
DRV_USART_CLIENT_STATUS type 1408
DRV_USART_CLIENTS_NUMBER macro 1362
DRV_USART_ClientStatus function 1384
DRV_USART_Close function 1383
drv_usart_config_template.h 1434
DRV_USART_COUNT macro 1427
DRV_USART_Deinitialize function 1376
DRV_USART_ERROR enumeration 1423
DRV_USART_ErrorGet function 1385
DRV_USART_INDEX macro 1363
DRV_USART_INDEX_0 macro 1412
DRV_USART_INDEX_1 macro 1413
DRV_USART_INDEX_2 macro 1414
DRV_USART_INDEX_3 macro 1415
DRV_USART_INDEX_4 macro 1416
DRV_USART_INDEX_5 macro 1417
DRV_USART_INIT type 1409
DRV_USART_INIT_FLAGS type 1410
DRV_USART_Initialize function 1374
DRV_USART_INSTANCES_NUMBER macro 1364
DRV_USART_INTERRUPT_MODE macro 1363
DRV_USART_INTERRUPT_SOURCE_ERROR macro 1364
DRV_USART_INTERRUPT_SOURCE_RECEIVE macro 1366
DRV_USART_INTERRUPT_SOURCE_RECEIVE_DMA macro 1366
DRV_USART_INTERRUPT_SOURCE_TRANSMIT macro 1366
DRV_USART_INTERRUPT_SOURCE_TRANSMIT_DMA macro 1367
DRV_USART_LINE_CONTROL enumeration 1430
DRV_USART_LINE_CONTROL_SET_RESULT enumeration 1424
DRV_USART_LineControlSet function 1388
DRV_USART_Open function 1381
DRV_USART_OPERATION_MODE enumeration 1431
DRV_USART_OPERATION_MODE_DATA union 1425
DRV_USART_PERIPHERAL_ID macro 1364
DRV_USART_QUEUE_DEPTH_COMBINED macro 1367
DRV_USART_Read function 1397
DRV_USART_READ_ERROR macro 1428
DRV_USART_READ_WRITE_MODEL_SUPPORT macro 1368
DRV_USART_ReadByte function 1401
DRV_USART_RECEIVE_DMA macro 1368
DRV_USART_ReceiverBufferIsEmpty function 1407
DRV_USART_ReceiverBufferSizeGet function 1404
DRV_USART_Status function 1377
DRV_USART_TasksError function 1380
DRV_USART_TasksReceive function 1378
DRV_USART_TasksTransmit function 1379
DRV_USART_TRANSFER_STATUS type 1411
DRV_USART_TransferStatus function 1405
DRV_USART_TRANSMIT_DMA macro 1369
DRV_USART_TransmitBufferIsFull function 1406
DRV_USART_TransmitBufferSizeGet function 1403
DRV_USART_Write function 1399
DRV_USART_WRITE_ERROR macro 1429
DRV_USART_WriteByte function 1402
drv_usb.h 1478
drv_usb_config_template.h 1479
DRV_USB_DEVICE_ENDPOINT_ALL macro 1477
DRV_USB_DEVICE_INTERFACE structure 1471
DRV_USB_EVENT enumeration 1468
DRV_USB_EVENT_CALLBACK type 1469
DRV_USB_HOST_INTERFACE structure 1473
DRV_USB_HOST_PIPE_HANDLE type 1470
DRV_USB_HOST_PIPE_HANDLE_INVALID macro 1467
DRV_USB_INDEX macro 1462
DRV_USB_INTERRUPT_SOURCE macro 1462
DRV_USB_OPMODE enumeration 1475
DRV_USB_PERIPHERAL_ID macro 1463
DRV_USB_ROOT_HUB_INTERFACE structure 1476
drv_wifi.h 1679
DRV_WIFI_ADHOC_CONNECT_ONLY enumeration member 1626
DRV_WIFI_ADHOC_CONNECT_THEN_START enumeration member 1626
DRV_WIFI_ADHOC_MODES enumeration 1626
DRV_WIFI_ADHOC_NETWORK_CONTEXT structure 1627
DRV_WIFI_ADHOC_START_ONLY enumeration member 1626
DRV_WIFI_AdhocContextSet function 1511
DRV_WIFI_BSSID_LENGTH macro 1583
DRV_WIFI_BssidGet function 1559
DRV_WIFI_BssidSet function 1527
DRV_WIFI_ChannelListGet function 1560
DRV_WIFI_ChannelListSet function 1506
DRV_WIFI_ConfigDataErase function 1555
DRV_WIFI_ConfigDataLoad function 1556
DRV_WIFI_ConfigDataPrint function 1557
DRV_WIFI_ConfigDataSave function 1558
DRV_WIFI_Connect function 1523
DRV_WIFI_ConnectContextGet function 1561

DRV_WIFI_CONNECTION_CONTEXT structure 1628
DRV_WIFI_CONNECTION_STATES enumeration 1629
DRV_WIFI_CONNECTION_TEMPORARILY_LOST 1485
DRV_WIFI_ConnectionStateGet function 1562
DRV_WIFI_ConnectStateMachine function 1505
DRV_WIFI_DEAUTH_REASONCODE_MASK macro 1584
DRV_WIFI_DEFAULT_ADHOC_BEACON_PERIOD macro 1585
DRV_WIFI_DEFAULT_ADHOC_HIDDEN_SSID macro 1586
DRV_WIFI_DEFAULT_ADHOC_MODE macro 1587
DRV_WIFI_DEFAULT_PS_DTIM_ENABLED macro 1588
DRV_WIFI_DEFAULT_PS_DTIM_INTERVAL macro 1589
DRV_WIFI_DEFAULT_PS_LISTEN_INTERVAL macro 1590
DRV_WIFI_DEFAULT_SCAN_COUNT macro 1591
DRV_WIFI_DEFAULT_SCAN_MAX_CHANNEL_TIME macro 1592
DRV_WIFI_DEFAULT_SCAN_MIN_CHANNEL_TIME macro 1593
DRV_WIFI_DEFAULT_SCAN_PROBE_DELAY macro 1594
DRV_WIFI_DEFAULT_SOFTAP_HIDDEN_SSID macro 1678
DRV_WIFI_DEFAULT_WEP_KEY_INDEX macro 1676
DRV_WIFI_DEFAULT_WEP_KEY_TYPE macro 1595
DRV_WIFI_Deinitialize function 1502
DRV_WIFI_DEVICE_INFO structure 1630
DRV_WIFI_DEVICE_TYPES enumeration 1631
DRV_WIFI_DeviceInfoGet function 1563
DRV_WIFI_DISABLED macro 1596
DRV_WIFI_DISASSOC_REASONCODE_MASK macro 1597
DRV_WIFI_Disconnect function 1524
DRV_WIFI_DOMAIN_CODES enumeration 1632
DRV_WIFI_EasyConfigTask_RtosTask function 1535
DRV_WIFI_ENABLED macro 1598
DRV_WIFI_EVENT_CONN_TEMP_LOST_CODES enumeration 1633
DRV_WIFI_EVENT_CONNECTION_FAILED 1486
DRV_WIFI_EVENT_INFO enumeration 1634
DRV_WIFI_EVENTS enumeration 1635
DRV_WIFI_GENERAL_ERRORS enumeration 1636
DRV_WIFI_GratuitousArpStart function 1525
DRV_WIFI_GratuitousArpStop function 1526
DRV_WIFI_HIBERNATE_STATES enumeration 1637
DRV_WIFI_HibernateEnable function 1517
DRV_WIFI_HWMulticastFilterGet function 1564
DRV_WIFI_HWMulticastFilterSet function 1520
DRV_WIFI_Initialize function 1503
DRV_WIFI_InitStateMachine_RtosTask function 1536
DRV_WIFI_INT_Handle function 1544
DRV_WIFI_isHibernateEnable function 1578
DRV_WIFI_ISR_RtosTask function 1537
DRV_WIFI_ISR_SemLock function 1545
DRV_WIFI_ISR_SemUnlock function 1538
DRV_WIFI_MAC_STATS structure 1638
DRV_WIFI_MacAddressGet function 1565
DRV_WIFI_MacAddressSet function 1530
DRV_WIFI_MACProcess_RtosTask function 1539
DRV_WIFI_MacStatsGet function 1566
DRV_WIFI_MAX_CHANNEL_LIST_LENGTH macro 1599
DRV_WIFI_MAX_NUM_RATES macro 1600
DRV_WIFI_MAX_SECURITY_KEY_LENGTH macro 1601
DRV_WIFI_MAX_SSID_LENGTH macro 1602
DRV_WIFI_MAX_WEP_KEY_LENGTH macro 1603
DRV_WIFI_MAX_WPA_PASS_PHRASE_LENGTH macro 1604
DRV_WIFI_MGMT_ERRORS enumeration 1640
DRV_WIFI_MGMT_INDICATE_SOFT_AP_EVENT structure 1642
DRV_WIFI_MIN_WPA_PASS_PHRASE_LENGTH macro 1605
DRV_WIFI_MRF24W_ISR function 1504
DRV_WIFI_MULTICAST_FILTER_IDS enumeration 1643
DRV_WIFI_MULTICAST_FILTERS enumeration 1644
DRV_WIFI_NETWORK_TYPE_ADHOC macro 1606
DRV_WIFI_NETWORK_TYPE_INFRASTRUCTURE macro 1607
DRV_WIFI_NETWORK_TYPE_P2P macro 1608
DRV_WIFI_NETWORK_TYPE_SOFT_AP macro 1609
DRV_WIFI_NetworkTypeGet function 1567
DRV_WIFI_NetworkTypeSet function 1507
DRV_WIFI_NO_ADDITIONAL_INFO macro 1610
DRV_WIFI_P2P_ERROR_CODES enumeration 1671
DRV_WIFI_P2P_STATES enumeration 1672
DRV_WIFI_POWER_SAVE_STATES enumeration 1645
DRV_WIFI_PowerSaveStateGet function 1568
DRV_WIFI_ProcessEvent function 1552
DRV_WIFI_PS_POLL_CONTEXT structure 1646
DRV_WIFI_PsPollDisable function 1518
DRV_WIFI_PsPollEnable function 1519
DRV_WIFI_REASON_CODES enumeration 1647
DRV_WIFI_RECONNECT_MODES enumeration 1648
DRV_WIFI_ReconnectModeGet function 1569
DRV_WIFI_ReconnectModeSet function 1508
DRV_WIFI_RegionalDomainGet function 1570
DRV_WIFI_RETRY_ADHOC macro 1611
DRV_WIFI_RETRY_FOREVER macro 1612
DRV_WIFI_RSSI_Cache_FromRxDataRead function 1542
DRV_WIFI_RSSI_Get_FromRxDataRead function 1543
DRV_WIFI_RssiGet function 1577
DRV_WIFI_RssiSet function 1534
DRV_WIFI_RTOS_TaskInit function 1546
DRV_WIFI_RTS_THRESHOLD_MAX macro 1613
DRV_WIFI_RtsThresholdGet function 1571
DRV_WIFI_RtsThresholdSet function 1531
DRV_WIFI_Scan function 1549
DRV_WIFI_SCAN_CONTEXT structure 1649
DRV_WIFI_SCAN_RESULT structure 1650
DRV_WIFI_SCAN_TYPES enumeration 1652
DRV_WIFI_ScanContextGet function 1551
DRV_WIFI_ScanContextSet function 1532
DRV_WIFI_ScanGetResult function 1550
DRV_WIFI_SECURITY_CONTEXT union 1653
DRV_WIFI_SECURITY_EAP macro 1614
DRV_WIFI_SECURITY_OPEN macro 1615
DRV_WIFI_SECURITY_WEP_104 macro 1616
DRV_WIFI_SECURITY_WEP_40 macro 1617
DRV_WIFI_SECURITY_WPA_AUTO_WITH_KEY macro 1618
DRV_WIFI_SECURITY_WPA_AUTO_WITH_PASS_PHRASE macro 1619
DRV_WIFI_SECURITY_WPS_PIN macro 1620
DRV_WIFI_SECURITY_WPS_PUSH_BUTTON macro 1621
DRV_WIFI_SecurityGet function 1572
DRV_WIFI_SecurityOpenSet function 1513
DRV_WIFI_SecurityTypeGet function 1579
DRV_WIFI_SecurityWepSet function 1514
DRV_WIFI_SecurityWpaSet function 1515

DRV_WIFI_SecurityWpsSet function 1516
DRV_WIFI_SetLinkDownThreshold function 1528
DRV_WIFI_SetPSK function 1512
DRV_WIFI_SOFT_AP_EVENT_REASON_CODES enumeration 1654
DRV_WIFI_SOFT_AP_STATES enumeration 1655
DRV_WIFI_SOFTAP_NETWORK_CONTEXT structure 1677
DRV_WIFI_SoftAPContextSet function 1554
DRV_WIFI_SoftApEventInfoGet function 1553
DRV_WIFI_SpiClose function 1547
DRV_WIFI_SpiInit function 1548
DRV_WIFI_SsidGet function 1573
DRV_WIFI_SsidSet function 1510
DRV_WIFI_STATUS_CODES enumeration 1656
DRV_WIFI_SWMULTICAST_CONFIG structure 1657
DRV_WIFI_SWMultiCastFilterEnable function 1529
DRV_WIFI_SWMulticastFilterSet function 1521
DRV_WIFI_TASK_MUTEX_Lock function 1540
DRV_WIFI_TASK_MUTEX_Unlock function 1541
DRV_WIFI_TX_MODES enumeration 1658
DRV_WIFI_TxModeGet function 1574
DRV_WIFI_TxModeSet function 1533
DRV_WIFI_TxPowerFactoryMaxGet function 1580
DRV_WIFI_TxPowerMaxGet function 1581
DRV_WIFI_TxPowerMaxSet function 1582
DRV_WIFI_WEP_CONTEXT structure 1659
DRV_WIFI_WEP_KEY_TYPE enumeration 1660
DRV_WIFI_WEP104_KEY_LENGTH macro 1622
DRV_WIFI_WEP40_KEY_LENGTH macro 1623
DRV_WIFI_WepKeyTypeGet function 1575
DRV_WIFI_WPA_CONTEXT structure 1661
DRV_WIFI_WPA_KEY_INFO structure 1662
DRV_WIFI_WPA_KEY_LENGTH macro 1624
DRV_WIFI_WPS_AUTH_TYPES enumeration 1663
DRV_WIFI_WPS_CONTEXT structure 1664
DRV_WIFI_WPS_CREDENTIAL structure 1665
DRV_WIFI_WPS_ENCODE_TYPES enumeration 1667
DRV_WIFI_WPS_ERROR_CONFIG_CODES enumeration 1673
DRV_WIFI_WPS_PIN_LENGTH macro 1625
DRV_WIFI_WPS_STATE_CODES enumeration 1674
DRV_WIFI_WPSCredentialsGet function 1576
drv_xc2c64a.h 381

E

EBI Driver Library 383
ENABLE_P2P_PRINTS macro 1668
ENABLE_WPS_PRINTS macro 1669
ENCx24J600 Driver Library Help 387
Endpoint Pipes 1444
Ethernet MAC Driver Library 425
Ethernet PHY Driver Library 463
Event Handling 1450
Example Code for Complete Operation 963
Example Usage of the Timer Driver 1263

F

File I/O Type Read Write/Data Transfer Model 1355
Files 18, 75, 94, 123, 199, 252, 304, 355, 381, 423, 460, 519, 649, 738, 833, 885, 943, 1002, 1061, 1136, 1244, 1339, 1432, 1478, 1679
ADC Driver Library 75
AK4384 Codec Driver Library 199
AK4642 Codec Driver Library 252
AK4645 Codec Driver Library 304
AK4953 Codec Driver Library 355
CPLD XC2C64A Driver Library 381
Driver Library 18
Ethernet MAC Driver Library 460
Ethernet PHY Driver Library 519
Graphics Driver Library 649
MRF24W Wi-Fi Driver Library 1679
NVM Driver Library 943
PMP Driver Library 1002
SD Card Driver Library 1061
SPI Driver Library 885, 1136
SPI Flash Driver Library 1244
Timer Driver Library 1339
USART Driver Library 1432
USB Driver Library 1478

G

GFX_CONFIG_OTM2201A_DRIVER_COUNT macro 535
GFX_CONFIG_S1D13517_DRIVER_COUNT macro 533
GFX_CONFIG_SSD1926_DRIVER_COUNT macro 534
GFX_CONFIG_TFT002_DRIVER_COUNT macro 536
GFX_PRIM_SetPIPWindow function 545
GFX_TCON_SSD1289Init function 589
Graphics Driver Library 523

H

How the Library Works 23, 98, 141, 207, 259, 311, 390, 660, 746, 849, 897, 958, 1019, 1067, 1142, 1254, 1348, 1439, 1483
ADC Driver 23
AK4384 Driver Library 141
AK4642 Driver Library 207
AK4645 Driver Library 259
AK4953 Driver Library 311
ENCx24J600 Driver 390
MRF24W Wi-Fi Driver Library 1483
MTCH6301 Driver Library 849
NVM Driver Library 897
PMP Driver Library 958
SD Card Driver Library 1019
SPI Driver Library 1067
SPI Flash Driver Library 1142
Timer Driver Library 1254
USART Driver Library 1348
USB Driver Library 1439

I

I2C Driver Library Help 657
I2C_DATA_TYPE type 725
I2C_SLAVE_ADDRESS_7bit type 726
I2C_SLAVE_ADDRESS_VALUE type 727
I2S Driver Library Help 742
Initialization 525
Initialization Overrides 1462
Initializing the Driver 850
Initializing the USART Driver 1349
Input Capture Driver Library 837

Introduction 3, 21, 79, 96, 128, 137, 204, 256, 308, 361, 365, 384, 388, 426, 464, 524, 658, 743, 838, 847, 895, 947, 956, 1006, 1017, 1065, 1140, 1252, 1344, 1437, 1481
- ADC Driver Library 21
- AK4384 Codec Driver Library 137
- AK4642 Codec Driver Library 204
- AK4645 Codec Driver Library 256
- AK4953 Codec Driver Library 308
- CAN Driver Library 128
- Comparator Driver Library 361
- CPLD XC2C64A Driver Library 365
- Driver Library 3
- EBI Driver Library 384
- ENCx24J600 Driver Library 388
- Ethernet MAC Driver Library 426
- Ethernet PHY Driver Library 464
- Graphics Driver Library 524
- Input Capture Driver Library 838
- MRF24W Wi-Fi Driver Library 1481
- MTCH6301 Driver Library 847
- NVM Driver Library 895
- Output Compare Driver Library 947
- OVM7690 Camera Driver Library 96
- PMP Driver Library 956
- RTCC Driver Library 1006
- SD Card Driver Library 1017
- SPI Driver Library 1065
- SPI Flash Driver Library 1140
- Timer Driver Library 1252
- USART Driver Library 1344
- USB Driver Library 1437

Isochronous Transfers 1458

L

LAYER_REGISTERS structure 587

Library Interface 6, 41, 80, 101, 129, 158, 221, 273, 322, 362, 369, 385, 396, 434, 471, 539, 674, 775, 839, 859, 910, 948, 968, 1007, 1027, 1090, 1168, 1270, 1371, 1466, 1494
- ADC Driver Library 41
- AK4384 Codec Driver Library 158
- AK4642 Codec Driver Library 221
- AK4645 Codec Driver Library 273
- AK4953 Codec Driver Library 322
- Camera Driver Library 101
- CAN Driver Library 129
- Comparator Driver Library 362
- CPLD XC2C64A Driver Library 369
- Driver Library 6
- EBI Driver Library 385
- Ethernet MAC Driver Library 434
- Ethernet PHY Driver Library 471
- Graphics Driver Library 539
- Input Capture Driver Library 839
- MRF24W Wi-Fi Library 1494
- NVM Driver Library 910
- Output Compare Driver Library 948
- PMP Driver Library 968
- RTCC Driver Library 1007
- SD Card Driver Library 1027
- SPI Driver Library 859, 1090
- SPI Flash Driver Library 1168
- Timer Driver Library 1270
- USART Driver Library 1371
- USB Driver Library 1466

Library Overview 22, 97, 140, 206, 258, 309, 366, 390, 429, 466, 525, 660, 746, 849, 897, 957, 1018, 1067, 1141, 1253, 1345, 1438, 1482
- ADC Driver Library 22
- AK4384 Driver Library 140
- AK4642 Driver Library 206
- AK4645 Driver Library 258
- AK4953 Driver Library 309
- CPLD XC2C64A Driver Library 366
- Ethernet MAC Driver Library 429
- Ethernet PHY Driver Library 466
- Graphics Driver Library 525
- MRF24W Wi-Fi Driver Library 1482
- MTCH6301 Driver Library 849
- NVM Driver Library 897
- PMP Driver Library 957
- SD Card Driver Library 1018
- SPI Driver Library 1067
- SPI Flash Driver Library 1141
- Timer Driver Library 1253
- USART Driver Library 1345
- USB Driver Library 1438

M

MAX_NONBUFFERED_BYTE_COUNT macro 999

Migrating Applications from v1.03.01 and Earlier Releases of MPLAB Harmony 889

MRF24W Wi-Fi Driver Library 1480

MTCH6301 Driver Library 846

N

NVM Driver Initialization 898

NVM Driver Library 888

O

Opening the Driver 851, 1144, 1442

Opening the USART Driver 1353

Operations 526

Optional Interfaces 1262

Others 1462

OTM2201A_TASK enumeration 626

Output Compare Driver Library 946

OVM7690 Camera Driver Library 96

P

Parallel Master Port (PMP) Driver Library 955

Period Modification 1258

PIP_BUFFER macro 564

Pipe Close 1446

Pipe Setup 1444

Pipe Stall 1447

Pipe Status 1445

Pipe Transfer Queue 1446

PMP_QUEUE_ELEMENT_OBJECT structure 1001

R

Rendering 526
RTCC Driver Library 1005

S

Sample Functionality 1490
SD Card Driver Initialization 1020
SDCARD_DETECTION_LOGIC enumeration 1050
SDCARD_MAX_LIMIT macro 1051
Secure Digital (SD) Card Driver Library 1016
SPI Driver Library 1064
SPI Flash Driver Libraries 1139
SST25FV016B API 1168
SST25VF020B API 1191
SST25VF064C API 1218
System Access 141, 207, 259, 311, 661, 747, 1068
System Initialization 24, 959, 1484
System Initialization and Deinitialization 1143
System Interaction 1255

T

Tasks Routine 853
Timer Driver Library 1251
Touch Input Read Request 852
Transfer Abort 1461
Transfer Operation 960
Transfer Requests 1452
Transfer Status 1459

U

USART Driver Library 1343
USB Driver Library 1436
Using the Library 22, 97, 137, 204, 256, 308, 366, 389, 427, 465, 525, 659, 745, 848, 896, 957, 1018, 1066, 1141, 1253, 1345, 1438, 1482
    ADC Driver Library 22
    AK4384 Codec Driver Library 137
    AK4642 Codec Driver Library 204
    AK4645 Codec Driver Library 256
    AK4953 Codec Driver Library 308
    CPLD XC2C64A Driver Library 366
    Ethernet MAC Driver Library 427
    Ethernet PHY Driver Library 465
    Graphics Driver Library 525
    MRF24W Wi-Fi Driver Library 1482
    MTCH6301 Driver Library 848
    NVM Driver Library 896
    PMP Driver Library 957
    SD Card Driver Library 1018
    SPI Driver Library 1066
    SPI Flash Driver Library 1141
    Timer Driver Library 1253
    USART Driver Library 1345
    USB Driver Library 1438
Using the USART Driver with DMA 1360

W

WF_WPS_PIN_LENGTH macro 1670