



MPLAB Harmony Help Template

MPLAB Harmony Integrated Software Framework

Introduction

This section introduces the MPLAB Harmony Core.

Description

The MPLAB Harmony development platform and ecosystem is based on a layered framework of interoperable library components. MPLAB Harmony Library Components are active and normally implement one or more independent, but cooperative state machines. These state machines service requests made by the application or other “client” modules through a set of interface (or API) functions. Different library modules serve different purposes: from core system services, to drivers for peripheral devices, to middleware layers supporting communication protocols and software capabilities.

This volume is a programmer reference that details the interfaces to the libraries that make up MPLAB Harmony Core and explains how to use the libraries individually to accomplish the tasks for which they were designed.

Where to Begin

Provides a list of resources to assist with deciding where to begin with MPLAB Harmony.

Description

The help documentation provides a comprehensive source of information on how to use and understand MPLAB Harmony. However, you don't need to read the entire document before you start working with MPLAB Harmony.

MPLAB Harmony Overview

MPLAB Harmony supports all of Microchip's 32-bit MCUs, with the exception of the PIC32MM family of devices (PIC32MX, PIC32MK, and PIC32MZ devices are supported). The PIC32MM family is supported by the MPLAB Code Configurator (MCC). Under MPLAB Harmony, differences between device families and among the devices within a family are hidden (abstracted) from the application by libraries that provide consistent interfaces, allowing the same application code to support multiple devices across multiple families. Device-specific differences for middleware libraries, drivers, and system services are managed by the MPLAB Harmony Configurator (MHC) and effectively hidden from the developer.

Middleware application demonstrations are provided for all major libraries (Bluetooth, Bootloader, Crypto, Drivers, Decoder, Graphics, System Services, and more). Example applications are also provided for all peripherals. These example and demonstration projects can serve as the foundation for the intended application by combining various projects into an approximation of the end application.

Prior to using MPLAB Harmony, it is recommended to review the Release Notes for any known issues. A PDF copy of the release notes is provided in the `<install-dir>/doc` folder of your installation.

New Users

If you are completely new to MPLAB Harmony, it is best to follow the Guided Tour.

More Experienced Users

If you are already somewhat familiar with the MPLAB Harmony installation and online resources and you want to jump right into a specific topic, you can use the links in the following table.

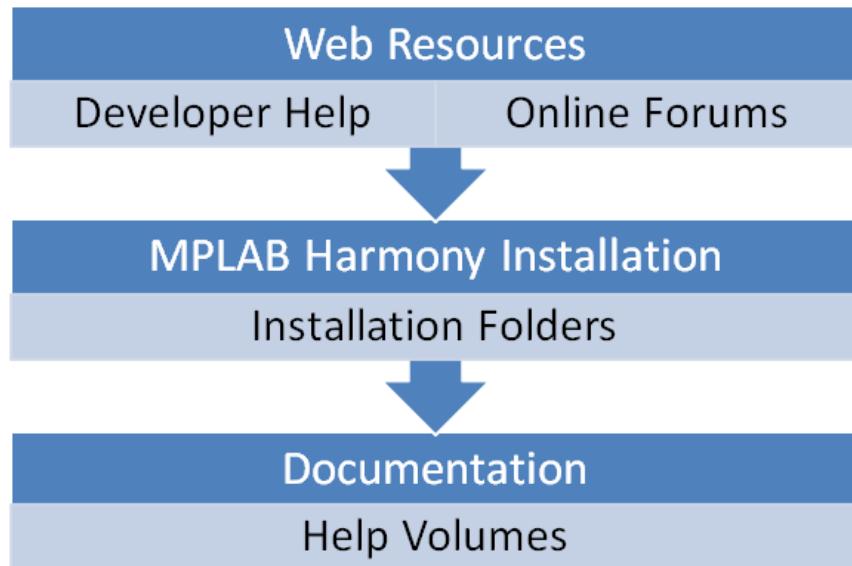
Start here...	If you...
Release Contents	...want to know what is included in this release.
Release Notes	...want to know what is new in this release and to learn of any known issues.
Prerequisites	...want to make sure that you have everything you need to begin working with MPLAB Harmony.
Using the Help	...want help using the documentation.
What is MPLAB Harmony?	...are new to MPLAB Harmony and need to understand the basic concepts.
MPLAB Harmony Development	...want to know how to develop MPLAB Harmony-compatible applications and libraries and how to best distribute and integrate them into an existing installation.
Porting and Updating to MPLAB Harmony	...are a MLA user and you need to port your application to MPLAB Harmony or you need information on updating an existing MPLAB Harmony project to a newer version of MPLAB Harmony.
Applications Help	...want to build and use the demonstration and example applications included in the installation.
MPLAB Harmony Framework Reference	...want to look up details on how to use the MPLAB Harmony framework libraries.

Third-Party Products Help	...need help with one of the third-party products included in this installation.
Board Support Package Help	...want to look up details on how to use MPLAB Harmony Board Support Packages.
Utilities Help	...need help using the MPLAB Harmony Configurator (MHC) or one of the other utilities included in this installation.
MHC Developer's Guide	...want information on how to create your own MPLAB Harmony library and integrated help.
MPLAB Harmony Compatibility Guide	...want to ensure software libraries you create are compatible with MPLAB Harmony.
Test Libraries Help	...want to know how to test your own MPLAB Harmony libraries.
Support	...need additional assistance.
Tips and Tricks	...want to learn more efficient and effective ways to use MPLAB Harmony.
Glossary	...need an explanation of the terms used in MPLAB Harmony.

Guided Tour

Provides a quick guided tour of the MPLAB Harmony installation and documentation and describes where to find additional information and help.

Description



Web Resources

This topic describes the main MPLAB Harmony Web site, from which you can download the installer and individual documents. In addition, it also describes where to find additional information and training, and how to locate the MPLAB Harmony online community.

Description

There are many Internet resources available for MPLAB Harmony, starting with the main MPLAB Harmony Web site: www.microchip.com/harmony.

This site contains introductory information and links to download the MPLAB Harmony installer and related documentation (also included in the installation). It also provides links to other resources you may require such as the MPLAB X IDE and XC32 language tools.

If you have not already done so, you can download the appropriate installer for your development workstation from the MPLAB Harmony Web site using the Downloads tab, as shown in the following figure.



The MPLAB Harmony installer is available for the Windows®, Linux®, and Mac® OS X platforms.

MPLAB® IDE

- Overview
- + MPLAB® X IDE
- MPLAB X IDE Debug Features by Device
- + MPLAB® XC Compilers
- Emulation Extension Pak
- + Emulator and Debugger Accessories
- Software Solutions Home
- + MPLAB Code Configurator
- + MPLAB Harmony
- + Microchip Libraries for Applications
- Additional Software Libraries
- Code Examples
- Embedded Code Source
- + MPLAB Xpress
- CAD/CAE Symbols
- SPICE Models
- MPLAB Mindi Analog Simulator
- Analog Simulation Files
- Other Software Libraries

Resources

- Training
- Data Sheets
- Support
- Sales
- Product Change Notification

MPLAB® Harmony Integrated Software Framework



MPLAB® Harmony is a flexible, abstracted, fully integrated firmware development platform for PIC32 microcontrollers. It takes key elements of modular and object oriented design, adds in the flexibility to use a Real-Time Operating System (RTOS) or work without one, and provides a framework of software modules that are easy to use, configurable for your specific needs, and that work together in complete harmony.

MPLAB Harmony includes a set of peripheral libraries, drivers and system services that are readily accessible for application development. The code development format allows for maximum re-use and reduces time to market.

MPLAB Harmony Features

- **Code Interoperability**
 - Modular architecture allows drivers and libraries to work together with minimal effort
- **Faster Time to Market**
 - Integrated single platform enables shorter development time
- **Improved Compatibility**
 - Scalable across PIC32 Microchip parts to custom fit customers requirement
- **Quicker Support**
 - One stop support for all customer needs including third party solutions
- **Easy third party software integration**
 - Integrates third party solutions (RTOS, Middleware, Drivers, etc.) into the software framework seamlessly

What's New in MPLAB Harmony V2:

- Optimized Peripheral Libraries (PLIBs)
- BSP Creator
- microMIPS Support
- Project Export
- App Templates
- Support for the Latest Compilers
- Updated Graphics Resources
 - New Graphics Library
 - Free Visual Graphic Design Tools
 - Display Manager for custom displays
 - Event Manager

Note: Some Web content is not shown to conserve space

Feature	Description
The Framework	MPLAB Harmony is a framework of system services, device drivers, and other libraries that are built upon a base of portable peripheral libraries to provide flexible, portable, and consistent software "building blocks" that you can use to develop your embedded PIC32 applications.



Refer to [The Microchip Web Site](#) for additional information.

Developer Help

Describes the Microchip Developer Help wiki site, which includes instructional videos and training.

Description

If you're new to MPLAB Harmony development, online training is available from the Microchip Developer Help site: microchip.wikidot.com/harmony:start. This site provides short introductory videos, self-paced training modules, and answers to frequently asked questions.

MICROCHIP Developer Help

Search This Site

Site updated 2 days ago
3245 active pages

Home Training Development Tools Functions

Get Started Here

- What is the MPLAB Harmony Framework?
- MPLAB Harmony Configurator (MHC)
- MPLAB Harmony Framework Overview
- MPLAB Harmony Libraries
- MPLAB Harmony Projects and Tutorials
- Advanced Software Framework v3 (ASF3)
- Microchip Libraries for Applications (MLA)
- Operating Systems
- Wi-Fi® and Ethernet
- Universal Serial Bus
- Wired Communications
- Wireless Communications
- Touch Sensing
- Displays
- Motor Control
- Power Conversion
- Signal Conditioning
- Digital Signal Processing
- Authentication
- Hardware-Software Integration

Projects Products Store Help

Home » MPLAB® Harmony

32-bit PIC® MICROCONTROLLERS **HARMONY** Integrated Software Framework

MPLAB® Harmony

The MPLAB® Harmony Integrated Software Framework is a flexible, abstracted, fully integrated firmware development platform for PIC32 microcontrollers. It takes key elements of modular and object oriented design, adds in the flexibility to use a Real-Time Operating System (RTOS) or work without one, and provides a framework of software modules that are easy to use, configurable for your specific needs, and that work together in complete harmony.

BE THE CONDUCTOR!
HARMONY Integrated Software Framework
FASTER PIC32 DEVELOPMENT WITH FEWER RESOURCES



Click image to enlarge.

Self-Paced Training

The material in these training modules exists elsewhere on this site in a general reference format. However, the training modules present it in an organized, step-by-step sequence to help you learn the topic from the ground up.

Tutorial / Class Title

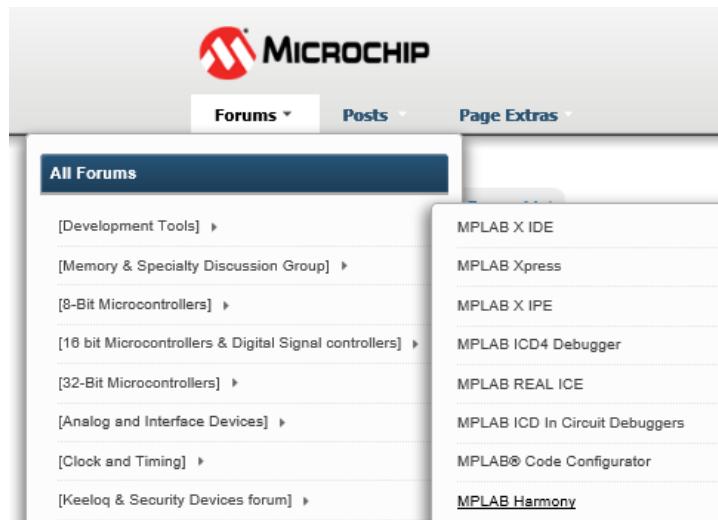
- MPLAB® Harmony v2.xx Graphics Library Training
- Creating Advanced Applications using MPLAB® Harmony
- Creating Simple Applications using MPLAB® Harmony
- Introduction to MPLAB® Harmony (Web-based)
- MPLAB® Harmony TCP/IP Stack Training
- MPLAB® Harmony Labs for ADC, UART, & USB Bootloader (chipKIT WF32)
- MPLAB® Harmony v1.xx Graphics Library Training
- MPLAB® Harmony Configurator (MHC) Tutorial Videos
- Introduction to MPLAB® Harmony (Videos)

Online Discussion Forum

Describes the MPLAB Harmony online community discussion forum.

Description

If you would like to interact with other MPLAB Harmony developers to share tips and tricks, you can sign onto the Microchip forums where you will find a forum dedicated to discussions about MPLAB Harmony. The Microchip Web Forums can be accessed online at: <http://www.microchip.com/forums>. From the Forums menu, select Forums > *Development Tools* > *MPLAB Harmony*.



Note: Refer to [Microchip Forums](#) for additional information.

Installation

Describes the contents and organization of the MPLAB Harmony installation.

Description

Default MPLAB Harmony Installation folders

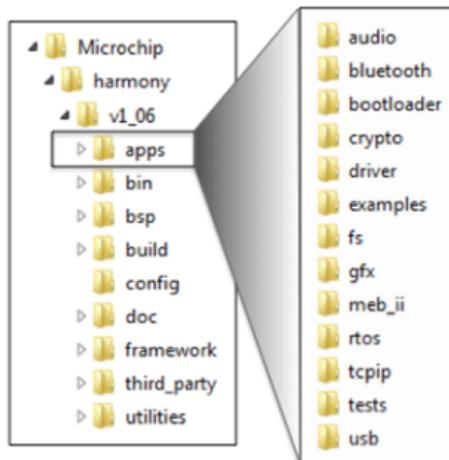
By default, MPLAB Harmony is installed into a version-specific folder.

- On Microsoft Windows Computers: C:\Microchip\harmony\<version>
- On Linux and Mac OS X Computers: ~/microchip/harmony/<version>

Where <version> is the version number of the installation. For example, v1_06.

Top-level Installation Folders

Within the main installation folder, the top-level folders organize the contents of the installation by type. It is best to become familiar with the organization of the installation because it is mirrored in the documentation, in the configuration tree, and elsewhere for consistency.



apps Folder

The `apps` folder contains application projects that demonstrate how to use various MPLAB Harmony libraries. Applications are grouped by type into sub-folders, as follows:

- Technology (bluetooth, bootloaders, tcpip, rtos, usb)
- Market (audio, crypto, gfx)
- MPLAB Harmony Layer (drivers, fs)
- Board (examples, meb_ii, tests)

You can use these projects to explore the capabilities of MPLAB Harmony and the supported demonstration and development boards. They also provide excellent examples of how to use the libraries for different purposes and you can use them as a starting point for your own projects.

bin Folder

The `bin` folder contains prebuilt binary (`.a`) files for some of the MPLAB Harmony libraries. Most libraries are provided as source code or generated from templates, which are provided in source form. However, there are a few libraries for which source code must be specially licensed. Also, the MPLAB Harmony peripheral libraries (PLIBs), while provided in source form, are also provided prebuilt at a high-level of optimization (`-O3`) so that users of the free version of the MPLAB XC32 C/C++ Compiler can take advantage of them.

bsp Folder

The `bsp` folder contains Board Support Packages (BSP) for the supported Microchip demonstration and development boards. An MPLAB Harmony BSP provides board-specific initialization and support code, configuration settings, and definitions that can be utilized by applications to more easily utilize the components provided on the selected board. Use of a BSP is not strictly required, because MPLAB Harmony libraries and applications can be configured without them. However, they are provided as a convenience to save the developer time tracing schematics and selecting configuration settings.

build Folder

The `build` folder provides MPLAB X IDE projects for the prebuilt libraries (in the `bin` folder) that are also provided in source form (like the PLIBs). This allows developers to modify settings such as optimization level or debug symbol support and rebuild the binary files, if desired.

config Folder

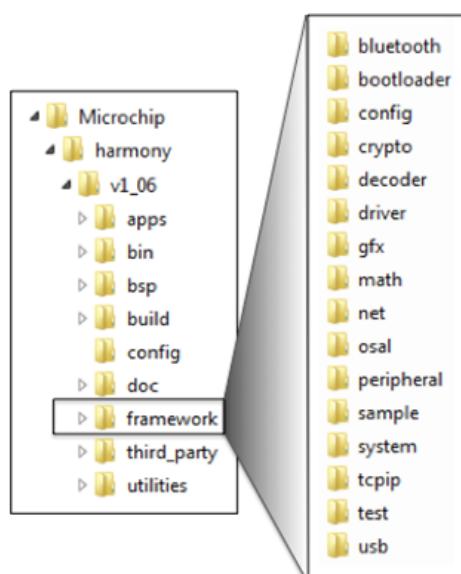
The `config` folder contains Hconfig configuration files for the MPLAB Harmony Configurator (MHC) graphical configuration utility used to simplify configuration of MPLAB Harmony projects. Hconfig files define the configuration options and menu choices for MPLAB Harmony libraries. They are linked together in a tree and you will find `config` folders and Hconfig files throughout the installation.

doc Folder

The `doc` folder contains the MPLAB Harmony Help documentation. It is provided in three formats: PDF, CHM, and HTML. The PDF format is an easily portable format, useful when viewing the documentation on a variety of devices and operating systems. When using this format, be sure to open your viewer's "Bookmarks" pane for easiest navigation. The CHM format is a popular Windows format that is useful when viewing the documentation on a Windows computer outside of the MPLAB X IDE. And, the MHC utilizes the HTML format (in the `html` subdirectory) to provide context-specific help from within the configuration tree. Refer to [Using the Help](#) for detailed information on each Help format.

framework Folder

The `framework` folder contains the MPLAB Harmony framework libraries, including all source code, interface header files, templates and configuration files.



Framework libraries are grouped in sub-folders by layer (drivers, system, peripheral, and osal) or by middleware library (bluetooth, crypto, decoder, gfx, math, tcpip, and usb) or by purpose (sample and test). Some of the groupings, particularly the layer groupings, are further broken down by peripheral or sub-system.

third_party Folder

The `third_party` folder groups all offerings provided by MPLAB Harmony third-party partners.

utilities Folder

The `utilities` folder contains development utilities, such as the Microchip MIB Compiler (`mib2bib`) and most notably, the MPLAB Harmony Configurator (MHC) plug-in for MPLAB X IDE.

Documentation

Describes the documentation provided in the MPLAB Harmony installation.

Description

As mentioned in the [Installation](#) section, the documentation is provided in the `doc` folder in three formats (PDF, CHM, and HTML). All three formats of the documentation provide the same content. The Help content is organized into the following seven volumes:

- Volume I: Getting Started With MPLAB Harmony
- Volume II: Supported Hardware
- Volume III: MPLAB Harmony Configurator (MHC)
- Volume IV: MPLAB Harmony Development
- Volume V: MPLAB Harmony Framework Reference
- Volume VI: Third-Party Products
- Volume VII: Utilities



Note: The individual Help volumes are only provided as PDF files. In addition to these files, a combined PDF that contains all six volumes, is also provided in your installation of MPLAB Harmony.

Volume I: Getting Started With MPLAB Harmony contains this brief guided tour, release information such as release contents and release notes, and information about how to get started using installing the MHC, and using applications.

Volume II: Supported Hardware contains detailed information on the Board Support Packages (BSP), as well as the hardware supported by MPLAB Harmony.

Volume III: MPLAB Harmony Configurator (MHC) contains the MPLAB Harmony Configurator User's guide, describing how to use the MHC, and the MPLAB Harmony Configurator Developer's Guide, describing how to develop new configuration files and templates to integrate new libraries into the MHC. It also contains the MPLAB Harmony Graphics Composer User's Guide, describing how to use the composer to create graphical user interfaces for your embedded devices.

Volume IV: MPLAB Harmony Development explains key MPLAB Harmony design concepts, provides information on porting existing libraries and applications to MPLAB Harmony, and lists general MPLAB Harmony development tips and tricks. It also contains guides for MPLAB Harmony compatibility, developing middleware and device drivers, and using the test harness library.

Volume V: MPLAB Harmony Framework Reference is the interface definition usage reference for all MPLAB Harmony framework libraries. It provides a complete API reference for every library, each of which contains the following sections.

- *Introduction* – A brief description of the library
- *Using the Library* – An overview of the library and description of its abstraction model, along with information on the common usage models for the library that include example code
- *Configuring the Library* – Information describing the libraries configuration and build options (the MHC's Help browser heavily references this section)
- *Library Interface* – The complete programmer's dictionary of interface functions, data types, and other definitions
- *Files* – A listing of the library's interface header files

When using a library for the first time, it is best to read through the **Introduction** and **Using the Library** sections to understand what the library does and how to use it. Then, when developing your own applications, refer to the **Library Interface** section for detailed information on function usage. The Help provides a convenient table of API functions, fully hyperlinked to complete descriptions for each.

The help sections in this volume for the libraries are grouped and organized in the same way that the source code for the libraries are organized under the framework folder within the installation.

Volume VI: Third-Party Products provides information on the third-party offerings included in the installation.

Volume VII: Utilities provides information on the development utilities provided in the installation, with the exception of the MHC, which has been provided in its own volume due to its significance to MPLAB Harmony.



Note: Refer to [Using the Help](#) for detailed information on the Help formats provided in MPLAB Harmony.

What is MPLAB Harmony?

This section provides an overview of MPLAB Harmony.

Description

Microchip MPLAB® Harmony is the result of a holistic, aggregate approach to creating firmware solutions for embedded systems using Microchip PIC32 microcontrollers. As shown in the following diagram, MPLAB Harmony consists of portable, modular and compatible libraries provided by Microchip and third-party ecosystem partners. MPLAB Harmony also includes easy-to-use development utilities like the MPLAB Harmony Configurator (MHC) plug-in for the MPLAB X IDE, which accelerate development of highly capable and reusable PIC32 embedded firmware applications.

MPLAB® Harmony Block Diagram

Designed almost completely in the C language (see **Note**), MPLAB Harmony takes key elements of modular and object-oriented design, adds in the flexibility to use a Real-Time Operating System (RTOS) or work without one if you prefer, and provides a framework of software modules that are easy to use, configurable for your specific needs, and that work together in complete harmony.



Note: MPLAB Harmony has not been tested with C++; therefore, support for this programming language is not provided.

Portability

Portability is a concern that is often overlooked when a silicon manufacturer provides software. However, breadth of solutions is a hallmark strength of Microchip, and MPLAB Harmony provides simple libraries to abstract away part-specific details and make a Microchip device easy to use, regardless of which device you choose. Any time you design a new product or update an existing one, cost must be balanced with capabilities; however, cost is more than just the bill of materials – it's also the Non-Refundable Engineering (NRE) cost to design and develop your solution. MPLAB Harmony provides peripheral libraries, device drivers, and other libraries that use clear and consistent interfaces, requiring little or no change in your application code and minimizing the engineering time and effort for each new design.

Device Drivers

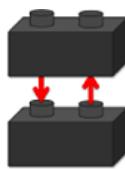
The primary purpose of a MPLAB Harmony device driver (or "driver") is to provide a simple and highly abstracted interface to a peripheral, allowing your application (or any other module in the system) to interact with a peripheral through a consistent set of functions. A driver is responsible for managing access to a peripheral, so that requests from different modules do not conflict with each other, and for managing the state of that peripheral so that it always operates correctly.

Peripheral Libraries

A Peripheral Library (PLIB) is a simple access library that provides a consistent (but very low level) interface to a peripheral that is "on board" the MCU. PLIBs hide register details, making it easier to write drivers that support multiple microcontroller families, but they are not normally used by applications directly to interact with peripherals, as they provide little abstraction, and because they require the caller to manage the detailed operation of a peripheral (including preventing conflicting requests from other modules). Because of the lack of conflict protection in a PLIB, only one module in a system should directly access the PLIB for a peripheral. Therefore, PLIBs are primarily used to implement device drivers (and some system services) to make them portable.

Modularity

MPLAB Harmony libraries are modular software "building blocks" that allow you to divide-and-conquer your firmware design. The interface to each library consists of a highly cohesive set of functions (not globally accessible variables or shared registers), so that each module can manage its own resources. If one module needs to use the resources of another module, it calls that module's interface functions to do so. Interfaces between modules are kept simple with minimal inter-dependencies so that modules are loosely coupled to each other. This approach helps to eliminate conflicts between modules and allows them to be more easily used together like building blocks to create the solutions you need.



Middleware Libraries

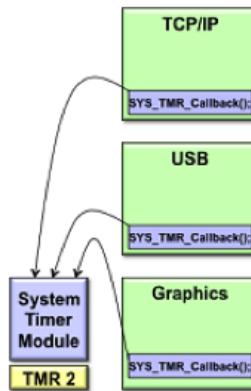
The normal usage models of some of the more complex peripherals, (i.e., USB or network interfaces) require interpreting complex protocols or may require substantial additional processing to produce useable results, such as drawing graphical images on an LCD screen with an LCD controller peripheral. Therefore, while a device driver may be completely sufficient for a simple peripheral like a UART, some peripherals require what is frequently called "middleware" (aptly named because it sits between your application and the hardware abstraction layer or "driver" layer). MPLAB Harmony provides several middleware library "stacks" to manage these more complex peripherals and provide the functionality you need and expect.

MPLAB Harmony middleware "stacks" are usually built upon device drivers and system services so that they can be supported on any Microchip microcontroller for which the required driver or service is supported. However, special purpose implementations may be available that integrate the driver, certain services, and various modules within the "stack" for efficiency.

System Services

MPLAB Harmony system services are responsible for managing shared resources so that other modules, such as drivers, middleware, and applications, do not conflict on shared resources. For example, if the TCP/IP, USB, and Graphics stacks attempted to concurrently use the Timer2 peripheral to perform some periodic task, they would very likely interfere with each other. However, if instead they used a timer system service (as the following image illustrates), it is the responsibility of the system service to keep the separate requests from interfering with each other. The timer service can be configured as desired for a specific system (for example, you may decide to use Timer3 instead of Timer2) isolating the necessary changes to the configuration of a single module and preventing potential conflicts.

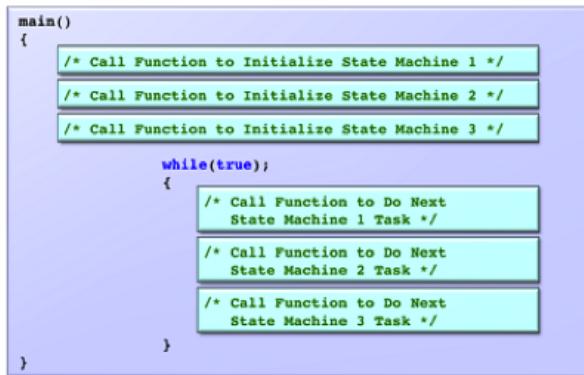
The use of a system service is very similar the use of a device driver, except that a driver normally requires the caller to "open" it to create a unique client-to-driver association. A system service does not normally require the caller to open the service before using it because system services are frequently shared by many clients within the system.



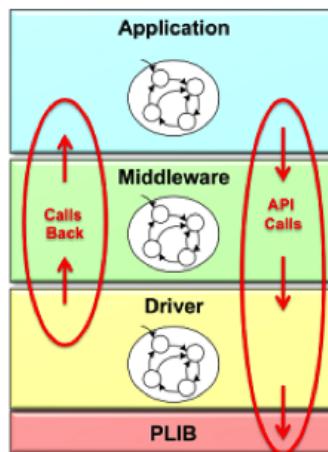
Compatibility

MPLAB Harmony modules (drivers, system services, and middleware – excluding PLIBs) are "active". This means when an application calls a module's interface function, the call will usually return immediately (unless a RTOS is in use) and the module will continue working on its own to complete the operation. Most modules will then provide a notification mechanism so the caller (i.e., client) can determine when the operation has finished.

Most MPLAB Harmony modules are implemented as cooperative state machines. The following image shows the basic idea of how this works. Each module has an "Initialize" function and each module has one (or more) "Tasks" function(s) to maintain its state machine(s). The state machines of all modules are initialized, shortly after the system comes out of reset in "main". After that (in a polled configuration, with no OS), the system drops into a "super loop" where each module's state machine function is repeatedly called, one after the other, to allow it to do the next "task" necessary to keep its state machine running. This allows the system to keep all modules running using a cooperative or shared "multi-tasking" technique. Modules (under control of your application) interact with each other by calling the interface functions of other modules (as illustrated in the following figure) and the system-wide "super loop" keeps all modules in the system running so they stay "active" and do their jobs.

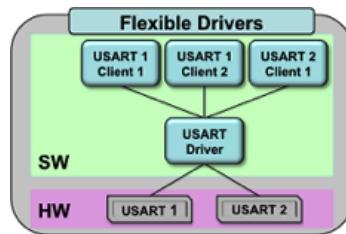


This method is not suitable for all needs; therefore, other configurations are possible. However, a polled configuration is the simplest to understand and it best illustrates the basic concept of how MPLAB Harmony allows independent modules to operate cooperatively within an embedded system. To interact with each other, otherwise independent library and application modules make calls to each other's Application Program Interface (API) functions, as shown in the following diagram. Calls *into* a library are made through well-defined API functions and calls back to the client may be made through *callback* functions, statically linked (at build time) or dynamically registered at run-time and called using a function pointer.



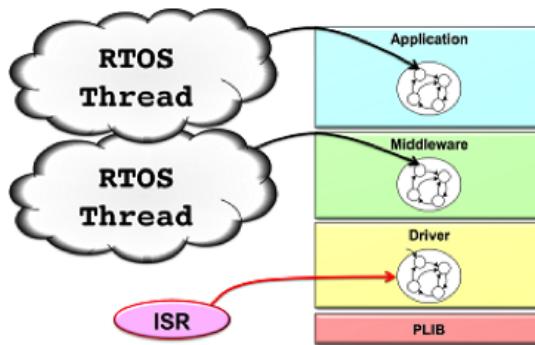
Flexibility

The basic MPLAB Harmony model of cooperating state machine driven modules, when combined with a little configurability, becomes flexible enough to meet the needs of almost any embedded system. For example, if you are using multiple identical peripherals, MPLAB Harmony has "dynamic" driver implementations that can manage all instances of a peripheral with a single instance of the driver code. You might also have a need for multiple "client" modules to use the same instance of a peripheral at the same time (such as the timer example, described previously). To manage this need, MPLAB Harmony has driver implementations that are intelligent enough to manage requests from multiple clients. On the other hand, your needs may be simpler than that. So, static and single client implementations are also available for key libraries to help reduce the amount of code and data storage needed for your system.



Or, your system may need to combine several middleware stacks and multiple, potentially independent, applications. If that is the case, the simple polling operation, using the "super loop" method frequently seen in simple embedded systems may not be sufficient. When you start adding more modules, it becomes more and more difficult to meet the timing requirements of all peripherals using a simple polled super loop.

Fortunately, MPLAB Harmony modules are written so that (where appropriate) their state machines can be run directly from an Interrupt Service Routine (ISR) or a RTOS thread. Using an ISR allows you to eliminate the latency of waiting for the execution of other modules in the loop to finish before a time-critical event is serviced, and it allows you to use the interrupt prioritization capabilities available on Microchip devices to ensure that your system responds to events in the real world in real-time.



Additionally, the ability to schedule and prioritize different tasks for different modules can be obtained for modules that are not associated with a specific processor interrupt (such as many middleware modules and your application) using a RTOS. In fact, that is one of the main reasons to use a RTOS. When your system becomes complex enough that you start struggling to meet your timing requirements using the super loop method, it's time to use a RTOS.

Fortunately, MPLAB Harmony module state machine functions can be called from a loop in a RTOS thread just as easily as they can be called from a polled "super loop" in a system without a RTOS. To allow this, modules are designed to be "thread safe" by calling semaphore, mutex, and critical section operations through an Operating System Abstraction Layer (OSAL). The OSAL provides a consistent set of functions to call, regardless of which RTOS is being used (or even if no RTOS is used). This method makes the choice of RTOS to use, if any, into a configuration option. MPLAB Harmony supports several OS and non-OS configurations and support for more operating systems is possible. All that is required is to implement the OSAL functions appropriately for the desired OS.

Configurability

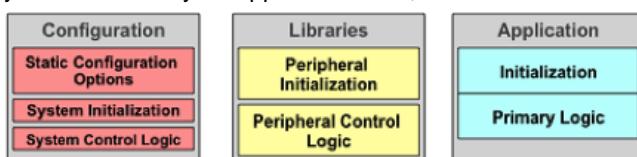
Most MPLAB Harmony libraries support a variety of build-time configuration options:

- Selection of the supported Microchip microcontroller
- Interrupt-driven or polled execution
- Static or Dynamic peripheral instance selection
- Single-client or Multi-client support
- Other library-specific options

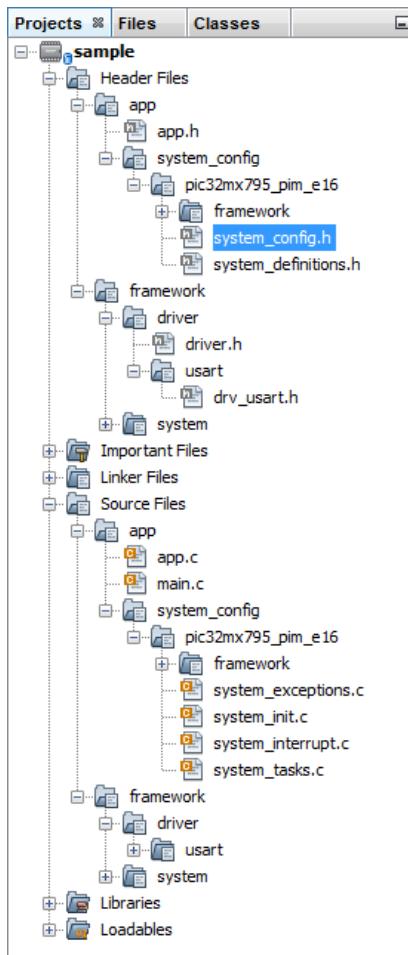
MPLAB Harmony libraries are designed to allow you to select a variety of configuration options to tailor them to your specific usage. For example, you may be able to select buffer sizes for data transfer modules or clock sources for timer modules. The set of configuration options for each library is identified and explained in the Help documentation (along with the interface and usage information) and the MPLAB Harmony Configurator (MHC) utility is provided to help simplify the process of configuring your system exactly the way you want and to get you started with a set of initial source files for your project.

Project Structure

To facilitate configurability, MPLAB Harmony projects are normally structured in a way that isolates the code necessary to configure a "system" from the library code and from your application code, as shown in the following figure.



The next figure shows how application, library, and configuration files are organized within the MPLAB X IDE project.



In a MPLAB Harmony project, the `main.c` file is kept very simple and consistent (containing primarily, just the super loop previously discussed). The application files (`app.c` and `app.h` in the previous figure) are separate from configuration files in the `system_config` sub-folders, so it is possible for a single application to have more than one configuration. (Usage of this capability can be seen in example and demonstration projects included with the installation of MPLAB Harmony.) The library modules that make up the MPLAB Harmony framework (in the `framework` folder) use the definitions provided in the selected configuration header (`system_config.h`, highlighted with a gray background in the previous figure) to specify the configuration options you selected when you configured the project. Finally the processor-specific peripheral libraries are provided as both a prebuilt binary (`.a` linker file) and as in-line source code to allow for maximum build efficiency for your firmware projects.

Summary

MPLAB Harmony provides a complete framework for developing your firmware solutions using Microchip microcontrollers and development tools. The firmware libraries and tools that make up the MPLAB Harmony framework are modular and compatible, making them simple to use. They're flexible and configurable, making them easy to tailor to your specific needs. And, they're portable across the full range of Microchip PIC32 microcontrollers, so you are sure to find a supported device that meets your needs.

The Main File

This topic describes the logic of the `main.c` file and the C language `main` function in a MPLAB Harmony project.

Description

The C language entry point for a MPLAB Harmony embedded application is the `main` function. This function is defined in the `main.c` file, generated in the project's `app` folder (or `src` directory on disk) by the MHC. The `main` function (see the following example) implements a simple "super loop", commonly used in embedded applications that do not make use of an operating system.

Example main Function Logic

```
int main ( void )
```

```

{
    /* Initialize all MPLAB Harmony modules, including application(s). */
    SYS_Initialize ( NULL );

    while ( true )
    {
        /* Maintain state machines of all polled MPLAB Harmony modules. */
        SYS_Tasks ( );
    }

    /* Execution should not come here during normal operation */
    return ( EXIT_FAILURE );
}

```

The SYS_Initialize Function

The first thing the main function does is to call a function named SYS_Initialize. The purpose of the SYS_Initialize function is to initialize every software module in the system. MPLAB Harmony is based upon a model of cooperating state machines. Therefore, this function must ensure that every module's state machine is placed in a valid initial state. The implementation of this function is one of the things generated by the MHC to configure a MPLAB Harmony system. This function's definition is generated in the `system_init.c` file, described in the [system_init.c](#) section.

 **Note:** The SYS_Initialize function signature has a "void *" input parameter. This is so that it may later be implemented in a library and an arbitrary initialization data structure may be passed into it. However, for a statically implemented SYS_Initialize function (which will normally be the case if you implement it yourself), this parameter is unnecessary and can be passed as "NULL".

The "Super Loop"

After all of the modules in the system have been initialized, the main function executes an infinite loop to keep the system running. This is commonly called a "super loop" as it is the outer-most loop, within which the entire system operates. This loop never exits. So, the code that exists after the end of that loop should never be executed and is only included there for safety, clarity, and syntactical completeness.

The SYS_Tasks File and the SYS_Tasks Function

Inside of the "super loop", the main function calls the SYS_Tasks function. The purpose of the SYS_Tasks function is to poll every module in the system to ensure that it continues to operate. This is how the system maintains the state machines of all polled modules. (Note that some modules may be interrupt driven and thus, not called from the SYS_Tasks function.) The implementation of the SYS_Tasks function is generated by the MHC in the `system_tasks.c` file, which is described in the [system_tasks.c](#) section.

The Application File(s)

This topic describes the normal structure of MPLAB Harmony application files.

Description

From the point of view of a MPLAB Harmony system, an application consists of two basic functions:

- APP_Initialize
- APP_Tasks

The application's initialization function (APP_Initialize) is normally called from the SYS_Initialize function, which is called from `main` before entering the top-level loop. The application's "tasks" function (APP_Tasks) is normally called from the SYS_Tasks function, which is called from `main` from inside the top-level loop. This is how the application's state machine is initialized and "polled" so that it can do its job. The SYS_Initialize function is normally implemented in the `system_init.c` file and the SYS_Tasks function is normally implemented in the `system_tasks.c` file. That is the convention for example and demonstration projects distributed with MPLAB Harmony and that is the case for projects generated by the MHC. You may do as you choose in your own projects, but it is recommended to follow this convention as it will make it easier to manage multiple configurations if you need them and it will be consistent with the MHC and other tools.

Application Initialization

An application's initialization function places the application's state machine in its initial state and may perform additional initialization if necessary. This function must not block and it should not call the routines of any other modules that may block. If

something needs to be initialized that may take time to complete, that initialization should be done in the application's state machine (i.e., in its "Tasks" function).

Sample Application Initialization Function:

```
void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state      = APP_STATE_INIT;
    appData.usartHandle = DRV_HANDLE_INVALID;
}
```

The sample project's initialization function initializes an internal variable and places the application's state machine in its initial state by assigning the APP_STATE_INIT enumeration value into the "state" member of the data structure that contains all of the data required by the application (appData). This structure is defined globally, but is only ever accessed by the application itself. The application's initialization function is called from the SYS_Initialize function (defined in `system_init.c`), which is called from `main` after a system Reset. Using this technique, the application is initialized (along with the rest of the system) whenever the system comes out of Reset.

Application Tasks

The application's state machine breaks up the job that the application must do into several short "tasks" that it can complete quickly, but between which it must wait for some other module to complete some tasks of its own. (In this case the other module is the USART driver.) Once each short task has completed successfully, the application transitions to another state to perform the next short task.

Example Application Tasks Function:

```
void APP_Tasks ( void )
{
    /* Handle returned by USART for buffer submitted */
    DRV_HANDLE usartBufferHandle;

    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Keep trying to open the driver until we succeed. */
        case APP_STATE_INIT:
        {
            /* open an instance of USART driver */
            appData.usartHandle = DRV_USART_Open(APP_UART_DRIVER_INDEX,
                                                DRV_IO_INTENT_WRITE);
            if (appData.usartHandle != DRV_HANDLE_INVALID)
            {
                /* Update the state */
                appData.state = APP_STATE_SEND_MESSAGE;
            }
            break;
        }

        /* Send the message when the driver is ready. */
        case APP_STATE_SEND_MESSAGE:
        {
            /* Submit message to USART */
            DRV_USART_BufferAddWrite(appData.usartHandle, &usartBufferHandle,
                                    APP_HELLO_STRING, strlen(APP_HELLO_STRING));
            if ( usartBufferHandle != DRV_HANDLE_INVALID )
            {
                /* Message is accepted. Driver will transmit. */
                appData.state = APP_STATE_IDLE;
            }
            break;
        }

        /* Idle state */
        case APP_STATE_IDLE:
        default:
        {

```

```
    /* Do nothing. */
    break;
}
}
```

Sample Application States

The sample application's "tasks" function breaks the operation of the application down in to the following states using a "switch" statement with the following "cases".

- APP_STATE_INIT
- APP_STATE_SEND_MESSAGE
- APP_STATE_IDLE

The sample application is placed into the APP_STATE_INIT state by the application's initialization function before the "tasks" function is ever called. So, the first time the APP_Tasks function is called, the switch statement executes the code under this case and the first short "task" the sample application attempts to do is open the USART driver to obtain a handle so that it can transfer data over the USART. Notice that the application checks the value of the handle returned from the [DRV_USART_Open](#) function to ensure that it is valid before it transitions to the APP_STATE_SEND_MESSAGE state. If the value of the handle returned be the driver's "open" function is invalid (equal to [DRV_HANDLE_INVALID](#)), the application stays in the APP_STATE_INIT state and continues trying to open the USART driver every time its "tasks" function is called. This technique allows a polled state machine to wait for something that it requires before continuing to avoid making inappropriate transitions to new states.

Once the application has a valid handle to the USART driver, it executes the code under the APP_STATE_SEND_MESSAGE case the next time its APP_Tasks function is called. In this state, the application calls a USART driver data transfer routine (DRV_USART_BufferAdd) to send the data data buffer string defined by the `system_config.h` header. Then, it checks the handle returned by the DRV_USART_BufferAddWrite function to see if it is valid. If the buffer handle is valid, it indicates that the USART driver has accepted the buffer and will take responsibility for the data transfer from that point forward. The application does not have to do anything else to cause the data transfer to occur. However, if the buffer is not accepted by the driver (in which case the handle returned by the DRV_USART_BufferAddWrite function would be invalid), the application stays in the APP_STATE_SEND_MESSAGE and tries again the next time the APP_Tasks function is called.

Once the application has successfully passed the buffer to the USART driver, it transitions to the APP_STATE_IDLE state where it stays and does nothing any time its "tasks" function is called. Its job is done! A more complex application would go on to some other task or potentially begin the process again. But, this is a simple "Hello World" sample application.



Note: The application is normally initialized last, after all other modules in the system have been initialized. But, it should never assume that any other module has completed its initialization when the application is initialized or when its "tasks" function is first called. Instead, it should always check the return value or status from any other module it calls to ensure that the call succeeded before moving on to the next state. Following this rule makes applications more robust and allows them to handle errors more effectively.

System Configurations

This section describes the files that make up a system configuration.

Description

In MPLAB Harmony, a system configuration consists of a set of files that define the build options, how the system is initialized, and how it runs after it has been initialized. The purpose of each of these files is described in the topics in this section.

`system_config.h`

This topic describes the purpose of system configuration header file.

Description

System Configuration

In MPLAB Harmony, most library modules require a set of build time configuration options that define a variety of parameters (such as buffer sizes, maximum or minimum limits, and default behavior). To configure a library for your specific needs, its configuration options can be defined using C language preprocessor `#define` statements. The set of configuration options

supported is described for each library in the "Configuring the Library" section of its help document and most libraries provide a template and example configuration header files in a `config` sub-folder within their `src` folder.

To obtain its build configuration options, every library includes the same common top-level configuration file that is named `system_config.h`, and it is generated by the MHC as part of your system configuration. The relative directory path to configuration directory that contains this file is defined in the build properties of your project configuration by the MHC so that the compiler can find it in its include file search path.

Example Configuration `system_config.h` Header

```

// ****
// ****
// Section: System Service Configuration
// ****
// ****

// ****
/* Common System Service Configuration Options
*/
#define SYS_VERSION_STR          "1.07"
#define SYS_VERSION              10700

// ****
/* Clock System Service Configuration Options
*/
#define SYS_CLK_FREQ              80000000ul
#define SYS_CLK_BUS_PERIPHERAL_1    80000000ul
#define SYS_CLK_UPLL_BEFORE_DIV2_FREQ 7999992ul
#define SYS_CLK_CONFIG_PRIMARY_XTAL 8000000ul
#define SYS_CLK_CONFIG_SECONDARY_XTAL 32768ul

/** Ports System Service Configuration ***/
#define SYS_PORT_AD1PCFG          ~0xffff
#define SYS_PORT_CNPUE              0x0
#define SYS_PORT_Cnen              0x0

// ****
// ****
// Section: Driver Configuration
// ****
// ****

// ****
/* USART Driver Configuration Options
*/
#define DRV_USART_INTERRUPT_MODE          false
#define DRV_USART_BYTE_MODEL_SUPPORT      false
#define DRV_USART_READ_WRITE_MODEL_SUPPORT true
#define DRV_USART_BUFFER_QUEUE_SUPPORT    true
#define DRV_USART_QUEUE_DEPTH_COMBINED    16
#define DRV_USART_CLIENTS_NUMBER          1
#define DRV_USART_SUPPORT_TRANSMIT_DMA   false
#define DRV_USART_SUPPORT_RECEIVE_DMA    false
#define DRV_USART_INSTANCES_NUMBER       1

#define DRV_USART_PERIPHERAL_ID_IDX0     USART_ID_2
#define DRV_USART_OPER_MODE_IDX0         DRV_USART_OPERATION_MODE_NORMAL
#define DRV_USART_OPER_MODE_DATA_IDX0    0x00
#define DRV_USART_INIT_FLAG_WAKE_ON_START_IDX0 false
#define DRV_USART_INIT_FLAG_AUTO_BAUD_IDX0 false
#define DRV_USART_INIT_FLAG_STOP_IN_IDLE_IDX0 false
#define DRV_USART_INIT_FLAGS_IDX0        0
#define DRV_USART_BRG_CLOCK_IDX0        80000000
#define DRV_USART_BAUD_RATE_IDX0        9600
#define DRV_USART_LINE_CNTRL_IDX0       DRV_USART_LINE_CONTROL_8NONE1
#define DRV_USART_HANDSHAKE_MODE_IDX0   DRV_USART_HANDSHAKE_NONE
#define DRV_USART_XMIT_INT_SRC_IDX0     INT_SOURCE_USART_2_TRANSMIT

```

```

#define DRV_USART_RCV_INT_SRC_IDX0           INT_SOURCE_USART_2_RECEIVE
#define DRV_USART_ERR_INT_SRC_IDX0           INT_SOURCE_USART_2_ERROR

#define DRV_USART_XMIT_QUEUE_SIZE_IDX0       10
#define DRV_USART_RCV_QUEUE_SIZE_IDX0       10

#define DRV_USART_POWER_STATE_IDX0          SYS_MODULE_POWER_RUN_FULL

// ****
// ****
// Section: Application Configuration
// ****
// ****

#define APP_UART_DRIVER_INDEX             DRV_USART_INDEX_0
#define APP_HELLO_STRING                 "Hello World\r\n"

```

The previous example defines configuration options for the application, system services, and USART driver used in the "pic32mx795_pim_e16" configuration of the sample project.

system_init.c

This topic describes the purpose of the system initialization file.

Description

In a MPLAB Harmony project, the `SYS_Initialization` function is called from the `main` function in order to initialize all modules in the system. This function is implemented as part of a system configuration by the MHC in a file named `system_init.c`. This file may also include other necessary global system items that must be implemented in order to initialize a system such as processor configuration bits and module initialization global data structures.

Example system_init.c File

```

// ****
// ****
// Section: Configuration Bits
// ****
// ****

/** DEVCFG0 **/

#pragma config DEBUG = OFF
#pragma config ICESEL = ICS_PGx2
#pragma config PWP = OFF
#pragma config BWP = OFF
#pragma config CP = OFF

/** DEVCFG1 **/

#pragma config FNOSC = PRIPLL
#pragma config FSOSCEN = OFF
#pragma config IESO = ON
#pragma config POSCMOD = XT
#pragma config OSCIOFNC = OFF
#pragma config FPBDIV = DIV_1
#pragma config FCKSM = CSDCMD
#pragma config WDTPS = PS1048576
#pragma config FWDTEN = OFF

/** DEVCFG2 **/

#pragma config FPLLIDIV = DIV_2
#pragma config FPLLMUL = MUL_20
#pragma config FPLLODIV = DIV_1
#pragma config UPLLIDIV = DIV_12
#pragma config UPLLEN = OFF

```

```
/** DEVCFG3 **/


#pragma config USERID = 0xfffff
#pragma config FSRSSEL = PRIORITY_7
#pragma config FMIEN = OFF
#pragma config FETHIO = OFF
#pragma config FCANIO = OFF
#pragma config FUSBIDIO = OFF
#pragma config FVBUSONIO = OFF



// ****
// **** Section: Driver Initialization Data
// ****
// ****



const DRV_USART_INIT drvUsart0InitData =

{
    .moduleInit.value = DRV_USART_POWER_STATE_IDX0,
    .uartID = DRV_USART_PERIPHERAL_ID_IDX0,
    .mode = DRV_USART_OPER_MODE_IDX0,
    .modeData.AddressedModeInit.address = DRV_USART_OPER_MODE_DATA_IDX0,
    .flags = DRV_USART_INIT_FLAGS_IDX0,
    .brgClock = DRV_USART_BRG_CLOCK_IDX0,
    .lineControl = DRV_USART_LINE_CNTRL_IDX0,
    .baud = DRV_USART_BAUD_RATE_IDX0,
    .handshake = DRV_USART_HANDSHAKE_MODE_IDX0,
    .interruptTransmit = DRV_USART_XMIT_INT_SRC_IDX0,
    .interruptReceive = DRV_USART_RCV_INT_SRC_IDX0,
    .queueSizeTransmit = DRV_USART_XMIT_QUEUE_SIZE_IDX0,
    .queueSizeReceive = DRV_USART_RCV_QUEUE_SIZE_IDX0,
};


// ****
// **** Section: Module Initialization Data
// ****
// ****



const SYS_DEVCON_INIT sysDevconInit =

{
    .moduleInit = {0},
};


// ****
// **** Section: System Data
// ****
// ****



/* Structure to hold the object handles for the modules in the system. */
SYSTEM_OBJECTS sysObj;



// ****
// **** Section: System Initialization
// ****
// ****



void SYS_Initialize ( void* data )
{
    /* Core Processor Initialization */
    SYS_CLK_Initialize( NULL );
```

```

sysObj.sysDevcon = SYS_DEVCON_Initialize(SYS_DEVCON_INDEX_0,
(SYS_MODULE_INIT*)&sysDevconInit);
SYS_DEVCON_PerformanceConfig(SYS_CLK_SystemFrequencyGet());
SYS_PORTS_Initialize();

/* Initialize Drivers */
sysObj.drvUsart0 = DRV_USART_Initialize(DRV_USART_INDEX_0, (SYS_MODULE_INIT
*)&drvUsart0InitData);

/* Initialize the Application */
APP_Initialize();
}

```

In addition to the SYS_Initialize function implementation, the previous example, the `system_init.c` file from the "pic32mx_795_pim_e16" configuration of the "sample" project, defines the processor configuration bits, data structures used to initialize the USART driver and device control service, and a global sysObj data structure used for the USART driver Device Control System Service returned by their initialization functions.

Note the SYSTEM_OBJECTS data type for the sysObj date structure is defined in the [system_definitions.h](#) section.

system_tasks.c

This topic describes the purpose of the system tasks file.

Description

Since MPLAB Harmony modules are state machine driven, they each have a "Tasks" function that must be called repeatedly (from the system-wide "super loop" in `main` or from an ISR or OS thread). The "Tasks" functions are all called from the top-level SYS_Initialize function that is normally implemented in a file called `system_tasks.c` that is generated by the MHC as part of a system configuration.

Example `system_tasks.c` File

```

void SYS_Tasks ( void )
{
    /* Maintain system services */
    SYS_DEVCON_Tasks(sysObj.sysDevcon);

    /* Maintain Device Drivers */
    DRV_USART_TasksTransmit(sysObj.drvUsart0);
    DRV_USART_TasksReceive(sysObj.drvUsart0);
    DRV_USART_TasksError (sysObj.drvUsart0);

    /* Maintain the application's state machine. */
    APP_Tasks();
}

```

The `system_tasks.c` file for the "pic32mx_795_pim_e16" configuration of the "sample" project, contains only the implementation of the SYS_Tasks function for that configuration. This function calls the tasks function of the Device Control System Service, the USART driver's tasks functions (it has three, one each for transmitter, receiver, and error-handling tasks), passing in the object handle returned from the driver's initialization routine, and it calls the application's tasks function APP_Tasks to keep the state machines of all three modules running.

system_interrupt.c

This topic describes the purpose of the system interrupts file.

Description

In an interrupt-driven configuration, any modules (such as drivers or system services) that can be driven from an interrupt must have their interrupt-capable tasks function(s) called from an Interrupt Service Routine (ISR) "vector" function instead of from the SYS_Tasks function. The form of the definition of the ISR vector function is dependent on what type of PIC32 microcontroller on which the system is running. So, any vector functions required are normally implemented as part of the a specific system configuration in a file normally named `system_interrupt.c`.

Since the sample application is entirely polled, its `system_interrupt.c` file does not contain any ISR vector functions. Refer to any interrupt-driven demonstration or example application to see how vector functions are implemented.

system_definitions.h

This topic describes the purpose of the system definitions header file.

Description

The system configuration source files (`system_init.c`, `system_tasks.c`, and `system_interrupt.c`) all require a definition of the system objects data structure and an `extern` declaration of it. The MHC generates these items in the `system_definitions.h` header file and the system source files all include that header file.

For example, the sample application defines the following structure definition and `extern` declaration.

```
typedef struct
{
SYS_MODULE_OBJ sysDevcon;
SYS_MODULE_OBJ drvUsart0;

} SYSTEM_OBJECTS;

extern SYSTEM_OBJECTS sysObj;
```

This structure holds the object handles returned by the Initialize functions for the device control and USART modules (in `system_init.c`) because they must be passed into the associated Tasks functions (called in `system_tasks.c`), which is why the system global `sysObj` structure requires an `extern` declaration. The MHC generates object handle variables in this structure for every instance of an active module in the system.

Additionally, the system configuration source files require the interface headers for all libraries and applications included in the system so that they have prototypes for their Initialize and Tasks functions. In the sample application, the `system_definitions.h` file includes the following interface headers (and standard C headers).

```
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include "system/common/sys_common.h"
#include "system/common/sys_module.h"
#include "system/clk/sys_clk.h"
#include "system/clk/sys_clk_static.h"
#include "system/devcon/sys_devcon.h"
#include "driver/usart/drv_usart.h"
#include "system/ports/sys_ports.h"
```



Note: The `system_configuration.h` header file should not be included by the application (`app.c`, `app.h`, or other) source files because it provides direct `extern` access to system objects and these objects should not be utilized by the application directly. The application (or other modules) should only interact with a module through its defined Application Program Interface (API), or client interface, not through the system objects or system functions that require that object.

system_exceptions.c

This topic describes the purpose of the system exceptions source file.

Description

The `system_exceptions.c` source file provides a skeletal implementation of the general exception handler function (shown below), overriding the weak function implementation provided by the MPLAB XC32 C/C++ Compiler that simply hangs in an endless loop.

```
void _general_exception_handler ( void )
{
    /* Mask off the ExcCode Field from the Cause Register.
    Refer to the MIPs Software User's manual. */
    _excep_code = (_CP0_GET_CAUSE() & 0x0000007C) >> 2;
    _excep_addr = _CP0_GET_EPC();
    _cause_str = cause[_excep_code];
```

```

SYS_DEBUG_PRINT(SYS_ERROR_ERROR,
"\nGeneral Exception %s (cause=%d, addr=%x).\n",
_cause_str, _excep_code, _excep_addr);

while (1)
{
    SYS_DEBUG_BreakPoint();
}
}

```

If a general exception occurs, this implementation will capture the address of the instruction that caused the exception in the `_excep_addr` variable, the cause code in the `_excep_code` variable and use a look-up table indexed by the cause to provide a debug message describing the exception if debug message support is enabled. Then, the function will hit a hard-coded debug breakpoint (in Debug mode) and hang in a loop to prevent runaway execution.

This implementation is provided to assist with development and debugging. The user is encouraged to modify this implementation to suit the needs of their system.

The use of this exception handler, instead of the compiler's built-in handler, is enabled in MHC by selecting the Options tab, and then selecting *Application Configuration > Exception Handling > Use MPLAB Harmony Exception Handler Template*. Advanced exception handlers, which provide more information about the exception, are available from the Options tab, by selecting *Advanced Exception and Error Handling*.

A detailed exploration of these options is discussed in Volume 1: Getting Started With MPLAB harmony Libraries and Applications > Creating Your First Project, Part II: Debugging With Your Project, and Part III: Debugging While Running Stand-alone.

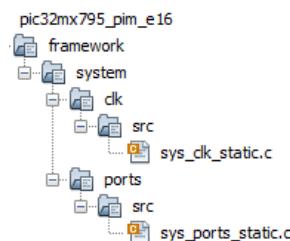
The Configuration-specific "framework" Folder

This topic describes the configuration-specific framework folder.

Description

The interface (i.e., API) headers and the source files for the dynamic implementations of all MPLAB Harmony libraries are contained in the main framework folder (`<install-dir>\framework`). However, the MHC generates any static implementations of the MPLAB Harmony libraries. These generated source files are all configuration specific because they are generated with knowledge of the configuration selections. Thus, they are placed in a configuration-specific framework folder. For consistency, the organization of the sub-folder tree of the configuration-specific framework folder matches the organization of the main framework folder.

For example, the configuration-specific framework folder for the `pic32mx795_pim_e16` configuration of the sample project contains the source files for the static, MHC-generated implementations of the Clock System Service and the Ports System Service, as shown in the following figure.



Other Configuration-specific Files

This topic describes two other (non C-language) configuration-specific files.

Description

There are two additional (non-C language) files generated by the MHC and placed into the configuration-specific folder. The first file, `<config-name>.mhc`, captures the configuration selections made by the user. The second file, which is always named `configuration.xml`, captures the various checksums and additional information required by the MHC to identify which generated files have been edited externally and to store miscellaneous other information it requires (such as the path to the MPLAB Harmony installation).

Driver Libraries Help

This section provides descriptions of the Driver libraries that are available in MPLAB Harmony.

Driver Library Overview

This topic provides help for the MPLAB Harmony driver libraries. It includes a general driver usage overview, as well as sections providing a programmer's reference for each driver that describes its interface and explains how to use it.

Drivers

Introduces MPLAB Harmony device drivers and explains common usage concepts.

Description

MPLAB Harmony device drivers (usually referred to as "drivers") provide simple, highly abstracted C-language interfaces to peripherals and other resources. A driver's interface allows applications and other client modules to easily interact with the peripheral it controls using consistent usage models. Some functions are similar on all drivers, while other functions are unique to a particular type of driver or peripheral. However, driver interface functions are generally independent of the details of how a given peripheral is implemented on any specific hardware or of how many instances of that peripheral exist in a given system.

Drivers normally utilize MPLAB Harmony Peripheral Libraries (PLIBs) to access and control peripheral hardware that is built into the processor (and is directly addressable by it). However, drivers can also support external peripheral hardware by calling another driver that directly controls a built-in peripheral to which the external peripheral is connected. For example, an SD Card driver may use a SPI driver to access its external SD Card Flash device. A driver may even be completely abstracted away from any hardware (utilizing no peripheral hardware at all), simply controlling some software resource (such as a buffer queue) or providing some service (such as data formatting or encryption). Using this method, driver and other modules may be "stacked" into layers of software, with each responsible for the details of managing its own resources while hiding those details from client modules that use them.

Regardless of the type of peripheral or resource that a MPLAB Harmony driver manages, a driver has the following fundamental responsibilities:

- Provide a common system-level interface to the resource
- Provide a highly abstracted file system style client interface to the resource
- Manage the state of the peripheral or resource
- Manage access to the resource

A driver's system interface can be thought of as being a horizontal interface and its client interface can be thought of as being a vertical interface, as shown in the following block diagram.

The horizontal or "system" interface provides functions to initialize the driver and keep it running. To keep a driver running, a system loop or ISR function (but never both in the same system) calls its state machine "tasks" function repeatedly, as necessary. Therefore, a driver's system interface is normally only called by code that is generated by the MPLAB Harmony Configurator (MHC) when you select and configure the driver. Its purpose is to ensure that the driver works independently (conceptually in the background), providing the capabilities it implements. By contrast, the application (or any other "client" of the driver) normally only interacts with the driver's vertical "client" interface (often thought of as the driver's API). The client interface provides functions to open the driver for use and interact with it, reading or writing data or performing device-type specific operations. The client interface is what allows the application to access the peripheral in a safe and easy way without worrying about the details of the driver or what other clients it may be serving.

The following sections describe in general terms how to use these two interfaces and give specific examples to help illustrate the concepts. The subsequent help sections for each individual driver describe their specific interfaces in detail; listing all supported functions, parameters, and return values as well as their data types and expected behavior. You may also refer to the MPLAB Harmony Driver Development guide for additional information on MPLAB Harmony drivers and for information on how to develop your own drivers, if needed.

Using a Driver's System Interface

Introduces the System Interface of a MPLAB Harmony device driver and explains its usage.

Description

An MPLAB Harmony driver's system interface provides functions to initialize, deinitialize, and reinitialize an instance of a driver, as well as functions to maintain its state machine (and/or implement its Interrupt Service Routine) and check its current "running" status. Normally, as an MPLAB Harmony application developer or a developer of a "client" module that uses the driver, you will not call the system interface functions directly. The MHC generates calls to the system interface functions of any driver that is used in a project when it generates the system configuration files. Exactly which functions are called and exactly how they're called depends on the configuration options selected in the project's active configuration.

For example, when the box next to "Use Timer Driver?" is selected in the MHC Options tree (within *MPLAB Harmony & Application Configuration > Harmony Framework Configuration > Drivers > Timer*), as shown in the following figure, the MHC will generate all necessary definitions and function calls for the Timer Driver's system interface.

Example Timer Driver MHC Options

These configuration selections, which are set by default once "Use Timer Driver" is selected, will cause the MHC to generate the following definitions in the `system_config.h` header file for the main project's current configuration when **Generate Code** is clicked.

Example Driver Options in `system_config.h`

```
/** Timer Driver Configuration ***/
#define DRV_TMR_INTERRUPT_MODE           true
#define DRV_TMR_INSTANCES_NUMBER        1
#define DRV_TMR_CLIENTS_NUMBER          1

/** Timer Driver 0 Configuration ***/
#define DRV_TMR_PERIPHERAL_ID_IDX0      TMR_ID_1
#define DRV_TMR_INTERRUPT_SOURCE_IDX0    INT_SOURCE_TIMER_1
#define DRV_TMR_CLOCK_SOURCE_IDX0       DRV_TMR_CLKSOURCE_INTERNAL
#define DRV_TMR_PRESCALE_IDX0           TMR_PRESCALE_VALUE_256
#define DRV_TMR_OPERATION_MODE_IDX0     DRV_TMR_OPERATION_MODE_16_BIT
#define DRV_TMR_ASYNC_WRITE_ENABLE_IDX0 false
#define DRV_TMR_POWER_STATE_IDX0        SYS_MODULE_POWER_RUN_FULL
```

It is important to notice that the Driver Implementation selection in the MHC graphical interface does not correlate to a `#define` statement in the `system_config.h` file. Instead, it determines which implementation of the driver this configuration will use. Drivers may have more than one implementation. For example, most drivers have both static and dynamic implementations. A static implementation is usually the smaller of the two, but it is only capable of controlling one instance of a peripheral. An equivalent dynamic implementation will be larger, but it is capable of managing multiple instances of the same type of peripheral using a single instance of the source code (and thus, one instance of the object code). Some drivers may have additional implementations, each one optimized for a different usage. The Driver Implementation pull-down control in the MHC graphical interface allows you to select which implementation the current configuration will use. Normally, you can use only a single implementation of a driver in a given configuration. If you change driver implementations, it changes which implementation is used for all instances of a peripheral.

The number of instances option, for example, Number of Timer Driver Instances, which correlates to the `DRV_TMR_INSTANCES_NUMBER` definition, determines how many instances of a static driver implementation will be generated or how many instances of a peripheral a dynamic driver implementation will manage. Drivers may also be designed to allow multiple different clients (applications or other modules) to share the same instance of a peripheral or resource. Therefore, a driver will have an option to determine a maximum number of simultaneous clients that it can support. For example, Number of Clients (`DRV_TMR_CLIENTS_NUMBER`) in the Timer Driver, which is fixed at one (1) and cannot be changed, which indicates that the Timer Driver is a single-client driver). The last implementation-specific configuration option in this example is the "Interrupt Mode" (`DRV_TMR_INTERRUPT_MODE`) setting. This option determines if the implementation is configured to run polled or interrupt driven (discussed further, in a following section). MPLAB Harmony drivers are generally designed to run most effectively in an interrupt-driven configuration, but they can also be run in a polled configuration to simplify debugging or to support task prioritization in an RTOS configuration.

The remaining configuration options are all instance-specific initialization options. For a dynamic implementation of a driver, these options are passed into the driver's Initialize function through an "init" data structure, as shown in the following example.

Example Driver Init Structure in `system_init.c`

```
const DRV_TMR_INIT drvTmr0InitData =
{
    .moduleInit.sys.powerState = DRV_TMR_POWER_STATE_IDX0,
    .tmrId = DRV_TMR_PERIPHERAL_ID_IDX0,
    .clockSource = DRV_TMR_CLOCK_SOURCE_IDX0,
    .prescale = DRV_TMR_PRESCALE_IDX0,
    .mode = DRV_TMR_OPERATION_MODE_16_BIT,
    .interruptSource = DRV_TMR_INTERRUPT_SOURCE_IDX0,
```

```

    .asyncWriteEnable = false,
} ;

```

The exact meaning and usage of these options are described in the **Configuring the Library** section in the Help documentation for each library. The live MHC Help windowpane displays the associated help section whenever you select one of these options in the options tree.

There is one instance-specific initialization option of which you should take special notice: the peripheral ID option (.tmrId, in the Timer Driver example shown). This initialization option associates the driver instance (a zero-based index number) with the peripheral-hardware instance number, as defined by the data sheet for the processor in use. For a dynamic driver, this association is actually made when the driver's initialize function is called and passes a pointer to the init data structure, as shown in the following code example.

Example Driver Initialize Call in system_init.c

```

/* Initialize Drivers */
sysObj.drvTmr0 = DRV_TMR_Initialize(DRV_TMR_INDEX_0, (SYS_MODULE_INIT *)&drvTmr0InitData);

```

In this example, the driver index (DRV_TMR_INDEX_0) is defined as a numeric constant with a value of zero (0). This line of code associates driver instance 0 with hardware timer instance 1 by calling the DRV_TMR_Initialize function from the system initialization code and passing a pointer to the `drvTmr0InitData` structure. As shown earlier, the Timer Driver's init structure contains the value TMR_ID_1 (defined by the timer peripheral library), in its `.tmrId` data member.

In a static implementation, the driver peripheral ID macro (DRV_TMR_PERIPHERAL_ID_IDX0) defined in `system_config.h` is hard-coded into the driver's instance-specific initialization function when it is generated by the MHC, instead of defining an "init" structure, as shown in the following example; however, the effect is the same.

Example Static Driver Initialize Function

```

void DRV_TMR0_Initialize(void)
{
    PLIB_TMR_Stop(DRV_TMR_PERIPHERAL_ID_IDX0);
    PLIB_TMR_ClockSourceSelect(DRV_TMR_PERIPHERAL_ID_IDX0, DRV_TMR_CLOCK_SOURCE_IDX0);
    PLIB_TMR_PrescaleSelect(DRV_TMR_PERIPHERAL_ID_IDX0, DRV_TMR_PRESCALE_IDX0);
    PLIB_TMR_Mode16BitEnable(DRV_TMR_PERIPHERAL_ID_IDX0);
    PLIB_TMR_Counter16BitClear(DRV_TMR_PERIPHERAL_ID_IDX0);
    PLIB_TMR_Period16BitSet(DRV_TMR_PERIPHERAL_ID_IDX0, 0);
}

```

The DRV_TMR0_Initialize function (with an instance number '0' in the name) in the previous example, is a static version of the DRV_TMR_Initialize system interface function. The call to this function is created by the MHC when it generates the system code. Therefore, that call is always generated with the correct name and with the correct instance number in the name. However, when calling client interface functions (open, close, read, write, etc.) from your own applications, you *should not* use an instance number in the function name. Dynamic drivers implement the client interface functions without any index numbers in their names. Instead, they use an index or handle parameter to identify the instance of the driver with which to interact. Also, when using static implementations of the drivers, the dynamic API functions are mapped (using the index or handle parameter) to the appropriate static function with the index number in its name. Therefore, calling the dynamic API function makes your application always portable, using whichever driver instance is configured to the index value with which you open the driver.



Note: Calling the static versions of the interface function (with the index numbers in their names) is not prohibited. However, it will limit the portability of your application.

Understanding this mechanism is critical to understanding how to access the desired peripheral hardware instance. Therefore, it is worth looking at a few demonstration applications to see how it is used. Also, refer to *Volume IV: MPLAB Harmony Development > Key Concepts > Key One-to-Many Relationships* for additional information on the concepts of having multiple implementations, instances, and clients.

Something else worth noting about the previous example call to the Timer Driver's initialize functions is that when using a dynamic implementation, it returns a value called an "object handle". In the previous example, that object handle was stored in a system configuration object data member (`sysObj.drvTmr0`). Object handles returned by module initialization functions are stored in a system configuration structure normally named `sysObj`. The definition of this structure is generated in the `system_definitions.h` header file the MHC, as shown in the following example.

Example System Object Data Structure Definition in system_definitions.h

```

typedef struct
{
    SYS_MODULE_OBJ    sysDevcon;
    SYS_MODULE_OBJ    drvTmr0;

} SYSTEM_OBJECTS;

extern SYSTEM_OBJECTS sysObj;

```

As shown in the previous example, this structure is “extern’d” for use by the other system files. It should not be used by application or library files, only by the system files for a single configuration. The `sysObj` structure is defined (and allocated in memory) by the `system_init.c` file, as shown in the following example.

Example System `sysObj` Definition in `system_init.c`

```
/* Structure to hold the object handles for the modules in the system. */
SYSTEM_OBJECTS sysObj;
```

For this discussion, you can ignore the `sysDevcon` member of the `SYSTEM_OBJECTS` structure as it will contain the handle for a different library. The important thing to note is that the `drvTmr0` member must be passed into the Timer Driver’s other system interface functions so that the driver has access to the data it needs manage that specific instance of itself (and the associated peripheral hardware), as shown by the following timer ISR example.

Example Timer ISR in `system_interrupt.c`

```
void __ISR(_TIMER_1_VECTOR, ip11AUTO) IntHandlerDrvTmrInstance0(void)
{
    DRV_TMR_Tasks(sysObj.drvTmr0);
}
```

In this ISR example, there are three important things to notice.

First, the ISR function itself is associated with a specific vector through the `__ISR` macro. Different interrupt vectors are associated with different peripheral instances and interrupts on different processors. That is why MPLAB Harmony ISR vector functions are generated in the configuration-specific `system_interrupt.c` file instead of being part of the driver library itself.

Second, the `DRV_TMR_Tasks` function implements the actual ISR logic of the TMR driver. Most MPLAB Harmony drivers are designed to run interrupt driven and their tasks functions implement the software state machine logic necessary to keep the driver’s interrupt sequence moving from one interrupt to the next until the driver’s task is complete.

Third, the `sysObj.drvTmr0` object handle’s value is passed into the driver’s tasks function so that it has access to the data it requires to control instance zero (0) of the Timer Driver and its associated hardware instance, which must match the ISR vector instance from which it is called.

By default, the Timer Driver is configured to run interrupt-driven, as shown previously. This is not necessarily true for all drivers. However, most drivers (including the Timer Driver) can run in a Polled mode by simply changing the configuration settings. For example, by clearing the “Interrupt Mode” option in the MHC configuration tree and regenerating the configuration code, the previous example ISR will be removed from `system_interrupt.c` and a call to the Timer Driver’s tasks function will be added to the polled system tasks function, as shown by the following `system_tasks.c` example code.

Example Call to Timer Tasks from `system_tasks.c`

```
void SYS_Tasks ( void )
{
    /* Maintain system services */
    SYS_DEVCON_Tasks(sysObj.sysDevcon);

    /* Maintain Device Drivers */
    DRV_TMR_Tasks(sysObj.drvTmr0);

    /* Maintain the application's state machine. */
    APP_Tasks();
}
```

In this example, the Timer Driver’s tasks function is called from the polled loop in main by the `SYS_Tasks` function. The driver’s tasks must still receive the `sysObj.drvTmr0` object handle value and its logic operates in exactly the same way, with one exception. Because the driver is now polled, the `DRV_TMR_INTERRUPT_MODE` option is now defined as false. This causes the driver to be built so that it does not enable its own interrupt, allowing it to run in the polled loop and to not require an ISR.

For additional information on the device driver system interface, refer to *Volume IV: MPLAB Harmony Development > MPLAB Harmony Driver Development Guide > System Interface* and to the documentation for the individual system interface functions for the driver in question.

Using a Driver's Client Interface

Introduces the Client Interface (or API) of a MPLAB Harmony device driver and explains common usage models.

Description

Applications (or any other “client” of a MPLAB Harmony device driver) normally only interact with the driver’s client interface (often called its API). The client interface provides functions to “open” the driver (creating a link between the client and the driver) and interact with it, to transfer data or perform operations that are specific to a given type of device, and to “close” the driver (releasing

the link). Once a driver has been configured and the configuration code has been generated, the application can assume that the driver will be initialized by the system-wide initialization function (SYS_Initialize) and that its tasks functions will be called as required from either the system-wide tasks function (SYS_Tasks) or from the appropriate ISR, depending upon how the driver was designed and configured.

To interact with the driver, a client must first call the driver's open function. This is necessary because all other client interface functions require a "handle" to the device driver that is returned by the open function, as shown in the following example.

Example Call to a Driver's Open Function

```
appData.handleTmr = DRV_TMR_Open(APP_TMR_DRV_INDEX, DRV_IO_INTENT_EXCLUSIVE);
if( DRV_HANDLE_INVALID != appData.handleTmr )
{
    // Advance to next application state.
}
```

In this example, the first parameter to the DRV_TMR_Open function is the APP_TMR_DRV_INDEX macro, which is a constant defined to the value of the desired driver instance index number in the `system_config.h` header file. This value must be the same as the index number used when the desired driver was initialized (as shown in the previous section). This is how the client becomes associated with a specific instance of a driver.

The second parameter identifies how the client intends to use the driver. Here, the client wants to have exclusive access to the driver. This means that no other client can currently have an active handle to this driver or this call will fail and return a value of `DRV_HANDLE_INVALID`. Drivers can also be opened as shared, as blocking or non-blocking and for reading, writing, or both.

Refer to the help for the `DRV_IO_INTENT` data type for additional information about the IO intent parameter of driver open functions. This parameter is merely an advisory parameter. How it is used by the driver is implementation dependent and will be described in the driver's help documentation.

Finally, if the open function was successful, the returned value will be a valid handle to the driver instance. This value is opaque and meaningless to the caller, but it must be passed back to the driver as the first parameter to every other client interface function provided by the driver. A valid handle identifies both the instance of the driver with which the caller interacts and it identifies the client performing the call. This means that, two different client applications or modules opening the same driver in the same system at the same time will receive different values for their "opened" handle. If, for any reason, the driver cannot support the "open" request (it is not finished initializing itself, it has already been opened for exclusive access, or cannot accept new open requests for any reason), it will return a value of `DRV_HANDLE_INVALID`, indicating the client cannot use it at this time. The `DRV_HANDLE_INVALID` value is the only non-opaque value that a client should consider meaningful. All other values are only meaningful to the driver that provided them.



Note: The `appData.handleTmr` variable in the previous example is a member of the application's `appData` structure. This structure is generated by the MHC as part of the initial application template and should be used to hold an applications state variables.

When the client is finished using a driver, it may close it, as shown in the following example.

Example Call to a Driver's Close Function

```
DRV_TMR_Close(appData.handleTmr);
```

This action releases the link to the driver, invalidating the handle and releasing any resources allocated by the driver to track requests from the client. Notice that the close function demonstrates the use of the driver handle, requiring it as a parameter. However, after the close function returns, the handle value cannot be used again. Therefore, the client should not call the driver's close function until it is done using the driver or it will have to call open again and obtain a new handle to use the driver again. In fact, since many embedded applications are always running, they often do not bother to close drivers they use. But, applications that can go idle or that can be stopped and restarted or that need to share a driver with other clients, but want to conserve resources, or that want use the driver exclusively, can close a driver when they are finished with it for a time and reopen it later when needed. In fact, this is a good way to share a single-client driver, or a driver that supports exclusive access, allowing each client to open it and use it only when a valid handle is obtained.

Using a Driver in an Application

Describes how to write a state-machine based application that uses a MPLAB Harmony driver.

Description

MPLAB Harmony generally treats all software modules, including applications, as state machines that have an "initialize" function and a "tasks" function. In fact, when not using a RTOS, it essentially treats the entire system as one large state machine that runs in a common super loop in the "main" function, as shown in the following code example.

Example Main Function

```

int main ( void )
{
    SYS_Initialize(NULL);

    while(true)
    {
        SYS_Tasks();
    }

    return (EXIT_FAILURE);
}

```

For the purpose of this discussion, it is important to understand that the application's APP_Initialize function is called from the SYS_Initialize function, along with the initialization of functions of all drivers and other libraries before execution enters the endless while(true) super loop that continuously calls the system-wide SYS_Tasks function. The application's APP_Tasks function is then called from the SYS_Tasks function inside of the super loop, along with all other polled modules in the system. If you are not already familiar with the organization of an MPLAB Harmony project, please refer to *Volume I: Getting Started With MPLAB Harmony > What is MPLAB Harmony?* for more information.

An application that uses a driver must define a **DRV_HANDLE** variable, as shown in the following example application header file.

Example Driver Application Header (app.h)

```

#include "driver/usart/drv_usart.h"

typedef enum
{
    APP_STATE_SETUP=0,
    APP_STATE_MESSAGE_SEND,
    APP_STATE_MESSAGE_WAIT,
    APP_STATE_DONE
} APP_STATES;

```

```

typedef struct
{
    APP_STATES    state;
    DRV_HANDLE    usart;
    char *        message;
} APP_DATA;

```

In this previous example, the driver handle variable is named usart. To keep the application well organized, it is common to keep all of the application's state variables (including one called "state" that holds the current state of the application's state machine) in a common structure (APP_DATA). This structure must be allocated in the application's source file (usually named `app.c`) and initialized by the application's initialization function, as shown in the following example.

Example Driver Application Initialization

```

APP_DATA appData;

void APP_Initialize ( void )
{
    /* Place the App in its initial state. */
    appData.state    = APP_STATE_SETUP;
    appData.usart    = DRV_HANDLE_INVALID;
    appData.message  = "Hello World\n";
}

```

The APP_Initialize function must initialize the state variable (appData.state) to put the application's state machine in its initial state (the APP_STATE_SETUP value from the APP_STATES enumeration). It must also initialize the driver-handle variable (appData.usart), so that the state machine knows it is not yet valid, and any other application variables (like the string pointer, appData.message).

Once the application's data structure has been initialized, it is safe for the system (the main and SYS_Tasks functions) to call the application's APP_Tasks function from the super loop to keep it running. The APP_Tasks function then executes state transition code as it switches between states, as demonstrated by the following example.

Example Application State Machine Using a Driver

```

void APP_Tasks ( void )
{
    switch ( appData.state )

```

```
}

case APP_STATE_SETUP:
{
    if (SetupApplication() == true)
    {
        appData.state = APP_STATE_MESSAGE_SEND;
    }
    break;
}

case APP_STATE_MESSAGE_SEND:
{
    if (MessageSend() == true)
    {
        appData.state = APP_STATE_MESSAGE_WAIT;
    }
    break;
}

case APP_STATE_MESSAGE_WAIT:
{
    if (MessageComplete() == true)
    {
        appData.state = APP_STATE_DONE;
    }
    break;
}

case APP_STATE_DONE:
default:
{
    break;
}
}
```

There are numerous ways to implement a state machine. However, in this example, the application changes state when the APP_Tasks function assigns a new value from the APP_STATES enumeration to the appData.states variable. This happens when one of the state transition function returns true. The end result is an overall application state machine execution that retries each state transition until it succeeds before moving on to the next state, as shown in the following diagram.

Application State Machine

 Note: The APP_STATE_ prefix and all inter-word underscores were removed from the state names to simplify the diagram.

After APP_Initialize places the state machine in its initial APP_STATE_SETUP state, the APP_Tasks function will call the SetupApplication function when it is called. When SetupApplication returns true indicating it has completed its task, the state machine advances to the next state. Otherwise, it stays in the same state and retries the tasks in the SetupApplication function. This pattern repeats for the APP_STATE_MESSAGE_SEND state and the MessageSend function as well as the APP_STATE_MESSAGE_WAIT state and the MessageComplete function. When all functions have returned true, the state machine transitions to the APP_STATE_DONE state where it unconditionally stays having completed its tasks.

The sum total of the tasks performed by each transition function completes the overall task of the application. For an application that uses a driver like this example, this includes opening the driver, sending the message, and closing the driver when the message has been sent. How each individual transition function in this example application accomplishes its portion of the overall task, is described in the examples in the following sections to demonstrate how drivers are commonly used.

Opening a Driver

Describes how to open a driver in a state-machine based application.

Description

To use a MPLAB Harmony driver, an application (or other client) must call the driver's "open" function and obtain a valid handle to it, as shown by the following code example.

Example Opening a Driver

```

static bool SetupApplication ( void )
{
    if (appData.usart == DRV_HANDLE_INVALID)
    {
        appData.usart = DRV_USART_Open(APP_USART_DRIVER_INDEX,
                                       (DRV_IO_INTENT_READWRITE | DRV_IO_INTENT_NONBLOCKING));
    }

    if (appData.usart == DRV_HANDLE_INVALID)
    {
        return false;
    }

    return true;
}

```

This example demonstrates the implementation of a state-transition function in a state machine-based application (as shown in the previous [Using a Driver in an Application](#) section). The `SetupApplication` function assumes that the `appData.usart` variable has been initialized to a value of `DRV_HANDLE_INVALID` when the application's state machine was initialized. Therefore, it checks this variable every time it is called to see if it has already completed its task. If `appData.usart` contains a value of `DRV_HANDLE_INVALID`, this indicates that the driver has not yet been successfully opened, causing the function to attempt to open the driver by calling `DRV_USART_Open`.

If the USART driver is ready and able to support a new client it will return a valid handle. If it is not ready or able to accept a new client, the driver will return `DRV_HANDLE_INVALID` and the `SetupApplication` function will return false and the application will stay in the same state and try to open the driver again the next time its state machine tasks function is called. When `DRV_USART_Open` returns a valid handle (a handle that is not equal to `DRV_HANDLE_INVALID`), the `SetupApplication` function returns true, allowing the application's state machine to advance.

This technique allows the application to try repeatedly to open the driver until it succeeds and guarantees that the application's state machine will not advance until it has done so. A more sophisticated application might use a time-out mechanism or some other error handling logic to take alternative action if it cannot open the driver in an acceptable time period. However, this simple implementation demonstrates the basic concept of how an MPLAB Harmony application (or any other client module) can safely open a driver before attempting to use it.

Using Driver Interface Functions

Describes how to use a device driver's synchronous client interface functions, such as those that read and write data.

Description

To use a MPLAB Harmony driver's client interface, the application must first obtain a valid handle from the driver's "open" function. The examples in this section assume that that has already occurred and that the value of the USART driver handle in the `appData.usart` variable is valid. The following example code demonstrates the implementation of a state transition function in a state machine-based application (as shown in the previous [Using a Driver in an Application](#) section) that writes data to a USART driver for transmission on the associated USART peripheral.

Example Writing Data To a Driver

```

static bool MessageSend ( void )
{
    size_t count;
    size_t length = strlen(appData.message);

    count = DRV_USART_Write(appData.usart, appData.message, length);

    appData.message += count;

    if (count == length)
    {
        return true;
    }

    return false;
}

```

In this example, the `appData.message` variable is a `char` pointer pointing to a null-terminated C-language string that was defined and initialized, as shown in the [Using a Driver in an Application](#) section. When `MessageSend` function is first called by the application's state machine, it points to the first character in the string to be transmitted. The function calculates the current length of the message string (using the standard C-language `strlen` function) and calls the driver's `DRV_USART_Write` function, passing it the valid driver handle (`appData.usart`) along with the pointer to the message string and its length, to transmit the message string on the associated USART.

If the driver is configured for blocking, the `DRV_USART_Write` function will not return until it has processed all of the data in the message string. However, that usually requires the use of a RTOS. Normally, in a bare-metal system (one that does not use a RTOS), MPLAB Harmony drivers are used in a non-blocking mode. In that case, a driver will perform as much of a task as it can when one of its interface functions is called without blocking. This means that the function will then return immediately, not waiting for the task to complete, and provide information on how much of the task was completed so the client can react appropriately. In this example, the `DRV_USART_Write` function will return a count of the number of bytes that were processed by the USART driver by this call to the function.

The `MessageSend` function captures the number of bytes processed by the `DRV_USART_Write` function in a local count variable. It then effectively removes those bytes from the message string by incrementing the pointer by count bytes (`appData.message` is a `char` pointer that increments by the size of one byte for every '1' added to it). Then, the `MessageSend` function checks to see if it was able to write the entire string by comparing the value of count to the value of length that it calculated before calling the driver's write function. If the two are equal, the task is complete and the `MessageSend` function returns true and the application's state machine can continue to the next state. If the two values are not equal, this indicates there are remaining bytes in the message string. The `MessageSend` function returns false and the application must stay in the same state so that the function can attempt to send the remaining bytes next time it is called. A driver only accepts data when it can process it; therefore, the client can call its data transfer function as many times as necessary, even when the function returns bytes processed if it cannot accept more data at that time.

When a client has called a driver interface function there are really only two possibilities. Either the operation has completed when the function returns, or the operation continues after the function has returned. If the operation completes immediately, the client can continue on without taking further action. However, in this example, while the USART driver may have accepted some of the bytes in the message string (perhaps copying them to an internal hardware or software FIFO buffer), it still takes some time to transmit the data over the USART peripheral. In many cases the client may need to know when the operation has actually completed. For this reason, most drivers provide one or more status functions that client applications may call to determine the current status of an operation, as demonstrated in the following example.

Example Using a Driver Status Function

```
static bool MessageComplete ( void )
{
    if (DRV_USART_ClientStatus(appData.usart) == DRV_USART_CLIENT_STATUS_BUSY)
    {
        return false;
    }
    return true;
}
```

This example extends the previous one and assumes that the `MessageSend` function has returned true and the application has moved to a new state where it calls this function to determine when the driver is idle, which indicates that the message has been completely transmitted. To do that, the `MessageComplete` function calls the `DRV_USART_ClientStatus` function. If its return value is `DRV_USART_CLIENT_STATUS_BUSY`, the USART driver is still working on a previous request by the client. If any other status value is returned, this indicates that the driver is no longer busy with a current request and the `MessageComplete` function returns true so that the client application's state machine can move on. A more sophisticated example would check for other possible status values that might indicate some error has occurred and take appropriate action. However, this example is sufficient to demonstrate the concept of checking a driver status function to determine when it is safe to move to another state.

Since the client application stays in the same state calling the status function each time its tasks function is called until the desired status is returned, it is effectively polling the status as if it were in a `while` loop. In fact, it is in the system-wide `while` loop. However, by not trapping the CPU within its own internal `while` loop, the application allows other modules (including, potentially, the driver it is using) to continue running and servicing requests. Failing to allow the rest of the system to run can result in a deadlock where the polling application is waiting for a status; however, the driver it is polling will never be able to provide the expected status, as the driver's own tasks function is not allowed to run. This is why it is important to use the technique described here to "poll" status from modules outside of the current module.

Using Asynchronous and Callback Functions

Describes how to use an asynchronous interface function to start a driver operation and receive a callback when the operation is

complete.

Description

When a client calls a function that is part of an asynchronous interface, the function starts the request and returns immediately, without finishing the request. The client can then either poll a status function to determine when the request has finished (as demonstrated in the Using Driver Interface Functions section) or it can utilize a callback function to receive a notification from the driver when the request has finished. So, the difference between an asynchronous interface and a synchronous interface is that a synchronous interface may finish all or part of the request before returning, whereas an asynchronous interface will always return immediately having only started the request. Determination of when the request has completed is handled separately.

The examples in this section reimplement some of the code from the example application described in the previous sections to demonstrate how to use asynchronous queuing and callback interfaces instead of the synchronous status-polling interface demonstrated in the Using Driver Interface Functions section. To use an asynchronous interface, we will first add a couple of new variables to our example application's data structure, as shown by the following structure definition.

Example Driver Application Header (app.h)

```
typedef struct
{
    APP_STATES          state;
    DRV_HANDLE          usart;
    char *              message;
    DRV_USART_BUFFER_HANDLE messageHandle;
    bool                messageDone;
} APP_DATA;
```

The `state`, `usart`, and `message` members of the `APP_DATA` structure are used in exactly the same way as they were in the previous examples. The `messageHandle` variable will be explained later and the `messageDone` variable is a Boolean flag used by the callback function to indicate to the application's state machine that the message has been completely processed by the driver. Using these new mechanisms results in very minor changes to the application's state machine, as shown in the following example `APP_Initialize` and `APP_Tasks` implementations.

Example Driver Application State Machine (app.c)

```
void APP_Initialize ( void )
{
    appData.state      = APP_STATE_SETUP;
    appData.usart      = DRV_HANDLE_INVALID;
    appData.message    = APP_MESSAGE;
    appData.messageHandle = DRV_USART_BUFFER_HANDLE_INVALID;
}

void APP_Tasks ( void )
{
    switch ( appData.state )
    {
        case APP_STATE_SETUP:
        {
            if (SetupApplication() == true)
            {
                appData.state = APP_STATE_MESSAGE_SEND;
            }
            break;
        }

        case APP_STATE_MESSAGE_SEND:
        {
            if (MessageSend() == true)
            {
                appData.state = APP_STATE_MESSAGE_WAIT;
            }
            break;
        }

        case APP_STATE_MESSAGE_WAIT:
        {
            if (appData.messageDone)
            {
                DRV_USART_Close(appData.usart);
            }
        }
    }
}
```

```
        appData.state = APP_STATE_DONE;
    }
    break;
}

case APP_STATE_DONE:
default:
{
    break;
}
}
```

As described previously, the `SetupApplication` state transition function opens the USART driver and the `MessageSend` function sends the message to it. However, there is no need for a `MessageComplete` state transition function. Instead, the application must implement a callback function that will set the `appData.messageDone` Boolean flag when the driver calls the application "back" to indicate that the message has been sent.

 **Note:** The AppInitialize function initializes the state, usart, and message members of the appData structure as previously described. And, it also initializes the messageHandle member with an invalid value to indicate that the message has not yet been sent. However, it does not initialize the messageDone flag because it is more appropriate to clear the flag elsewhere, immediately before calling the driver to send the message.

To use a callback mechanism requires the client to implement and register a callback function. A client must register this function after opening the driver, but prior to calling the driver to initiate the operation. This is often done in the same state transition that opens the driver, as shown in the following `SetupApplication` example.

Example Registering a Driver Callback Function

```

static void BufferDone ( DRV_USART_BUFFER_EVENT event,
                        DRV_USART_BUFFER_HANDLE bufferHandle,
                        uintptr_t context )
{
    APP_DATA *pAppData = (APP_DATA *)context;

    if (event == DRV_USART_BUFFER_EVENT_COMPLETE)
    {
        if (bufferHandle == pAppData->messageHandle)
        {
            pAppData->messageDone = true;
            return;
        }
    }

    /* Error */
    return;
}

static bool SetupApplication ( void )
{
    if (appData.usart == DRV_HANDLE_INVALID)
    {
        appData.usart = DRV_USART_Open(APP_USART_DRIVER_INDEX,
                                       (DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_NONBLOCKING));
    }

    if (appData.usart == DRV_HANDLE_INVALID)
    {
        return false;
    }

    DRV_USART_BufferEventHandlerSet(appData.usart, BufferDone, (uintptr_t)&appData);
    return true;
}

```

This code block implements both the `BufferDone` callback function and the application's `SetupApplication` state transition function. After successfully opening the driver, the `SetupApplication` function calls the `DRV_USART_BufferEventHandlerSet` function and passes it the driver handle (`appData.usart`) once it is valid, along with the address of the `BufferDone` callback.

function and a context value.

The context value can be anything that will fit in an integer large enough to hold a pointer (it is a `uintptr_t` variable). However, this parameter is most commonly used to pass a pointer to the caller's own data structure as demonstrated here (even though it is not strictly necessary). This is done primarily to support multi-instance clients. (Refer to *Volume IV: MPLAB Harmony Development > Key Concepts* for information on multiple instances.) A multi-instance client is designed to manage multiple instances of itself by allocating multiple instances of its own data structure, but only one instance of its object code. Passing a pointer to the data structure in the context variable identifies the specific instance that was used when calling the driver.

Once the callback function has been registered with the driver, the application can transition to a state where it attempts to initiate an asynchronous operation. The following example demonstrates the use of a buffer-queuing write function to transmit a message over the USART.

Example Queuing a Buffer to a Driver

```
static bool MessageSend ( void )
{
    appData.messageDone = false;
    DRV_USART_BufferAddWrite(appData.usart, &appData.messageHandle,
                            appData.message, strlen(appData.message));

    if (appData.messageHandle == DRV_USART_BUFFER_HANDLE_INVALID)
    {
        return false;
    }

    return true;
}
```

Before attempting to send the message, this implementation of the `MessageSend` state transition function clears the `appData.messageDone` flag so it can detect when the message has completed. Then, it calls the `DRV_USART_BufferAddWrite` function to queue up the buffer containing the message to be transmitted by the USART driver. To that function, it passes the USART driver handle (`appData.usart`), the address of the `appData.messageHandle` variable, the pointer to the message buffer (`appData.message`), and the size of the buffer in bytes as calculated by the `strlen` function. The USART driver then adds this buffer to its internal queue of buffers to transmit and provides a handle to the caller that identifies that buffer's place in the queue by storing it to the `appData.messageHandle` variable.

If, for some reason, the driver is unable to successfully queue up the buffer (perhaps the queue is full), it will assign a value of `DRV_USART_BUFFER_HANDLE_INVALID` to the `appData.messageHandle` variable. If that happens, the `MessageSend` function returns `false` and the application will stay in the same state and retry the operation again next time its tasks function is called. But, if the operation succeeds, the application advances to the next state.

Once the driver completes the operation, it will call the client's callback function. As shown in the `BufferDone` code example, the driver passes it an enumeration value that identifies which event has just occurred (the `DRV_USART_BUFFER_EVENT_COMPLETE` value) in the `event` parameter. It also passes it the handle of the buffer that has just completed (`bufferHandle`). The client can use the `bufferHandle` value to verify that it matches the value stored in the `appData.bufferHandle` variable to uniquely identify an individual buffer. This is very useful when a client queues up multiple buffers at the same, which is being shown in this example as a demonstration.

The context parameter to the `BufferDone` function contains a pointer to the application's global (`appData`) data structure. (This is the same value that was passed in the context parameter to the `DRV_USART_BufferEventHandlerSet` function.) While not strictly necessary in this example, it is very useful for multi-instance clients such as dynamic device drivers and middleware to identify which instance of the client requested the operation. The callback function simply casts the context value back into a pointer to the client's own data structure's data type (`APP_DATA` in this example) and uses it to access the structure members. (Again, please refer to *Volume IV: MPLAB Harmony Development > Key Concepts* for information on multiple instances.)

The callback function uses the `event` parameter to identify why the callback occurred. If it was called to indicate that the buffer has been processed, the `event` parameter will contain the value `DRV_USART_BUFFER_EVENT_COMPLETE`. If it contains any other value an error has occurred. The `BufferDone` callback also checks to verify that the buffer that completed was the same buffer that it queued up by comparing the `bufferHandle` value it was passed with the value assigned to the `appData.messageHandle` variable when the application called `DRV_USART_BufferAddWrite`. It accesses the message handle value it saved using the `pAppData` pointer given to it through the `context` parameter just. Once it has verified that the buffer it queued has completed, it sets the `pAppData->messageDone` flag to notify the application's state machine and execution returns to the driver.



Note: It is important to understand that the `MessageDone` callback function executes in the context of the driver, not the application. Depending on how the system is configured, this means that it may be called from within the driver's ISR context or from another thread context if using a RTOS.

In this example, the APP_Tasks application state machine function is essentially the same as the state machine for the synchronous example. The only difference is that when the application is in the APP_STATE_MESSAGE_WAIT state, it checks the `appData.messageDone` flag to determine when to close the driver and transition to the APP_STATE_DONE state instead of calling a transition function. (It could still do this in a state transition function, but it was done differently in this example to emphasize the concept.)

The advantage of using an asynchronous interface over a synchronous one is that it allows the client's state machine to continue on, potentially doing something else while the requested operation completes. Whereas a synchronous interface has the possibility of blocking the client's state machine until the operation finishes (when used in a RTOS configuration). An asynchronous interface will always return immediately without blocking (whether a RTOS is used or not). Because of this, most asynchronous interfaces will also allow queuing of more than one operation at a time. This allows client applications to keep a driver continuously busy by keeping the driver's queue full, maximizing data throughput or operation speed. By contrast, a synchronous interface requires one operation to complete before the synchronous function can be called again to cause the next one to begin.

The cost of this capability is that an asynchronous interface has the added complexity of a callback function (if the client cares when the operation finishes) and the fact that a callback function may be called from within the driver's ISR context, depending on how the driver was designed and configured. This fact generally restricts what can be done within the callback function. For example, it is usually a bad idea to perform lengthy processing within a callback function as it will block all lower priority ISRs (as well as the main loop or other threads) while that processing occurs. Also, it is usually best to not call back into the driver's own interface functions unless those functions are documented as being safe to call from within the driver's callback context. Many interface functions (particularly data transfer and data queuing functions) must use semaphores or mutexes to protect their internal data structures in RTOS environments and those constructs cannot be used from within an ISR.

It is also important to not make non-atomic (read-modify-write) accesses to the client's own state data from within the callback function, as the client cannot protect itself against an interrupt that is owned by the driver. That is why a separate Boolean flag variable is commonly used to indicate to the client that the callback has occurred. Most other processing should occur in the client's state machine. It is usually best to simply capture the event and return as quickly as possible from the callback function and let the application's state machine tasks function perform any lengthy processing or calling back into the driver.

Please refer to *Volume IV: MPLAB Harmony Development* for additional information.

Library Interface

Data Types

Constants

Files

Files

Name	Description
driver.h	Driver layer data types and definitions.

Description

driver.h

Driver layer data types and definitions.

Description

Driver Layer Interface Header

This file defines the common macros and definitions for the driver layer modules.

Remarks

The parent directory to the "driver" directory should be added to the compiler's search path for header files such that the following include statement will successfully include this file.

```
#include "driver/driver.h"
```

File Name

driver.h

Company

Microchip Technology Inc.

AT24 Driver Library Help

This section describes the AT24 External EEPROM Driver Library.

Introduction

This library provides an interface to access the external AT24 EEPROM over the I2C interface.

Description

This library provides a non-blocking interface to read and write to the external AT24 EEPROM. The library uses the I2C (or TWIHS) PLIB to interface with the AT24 EEPROM.

Key Features:

- Supports a single instance of the AT24 EEPROM and a single client to the driver.
- Supports page writes.
- Supports writes to random memory address and across page boundaries.
- The library can be used in both bare-metal and RTOS environments.

Using the Library

This topic describes the basic architecture of the AT24 Library and provides information on how to use the library.

Description

The AT24 library provides non-blocking APIs to read and write to external AT24 EEPROM. It uses the I2C (or TWIHS) peripheral library to interface with the AT24 EEPROM.

- The library provides APIs to perform reads/writes from/to any EEPROM memory address, with number of bytes spanning multiple pages.
- The library provides API to perform page write to EEPROM. Here, the memory start address must be aligned to the EEPROM page boundary.
- Application can either register a callback to get notified once the data transfer is complete or can poll the status of the data transfer.
- The library can be used in both bare-metal and RTOS environments.

Abstraction Model

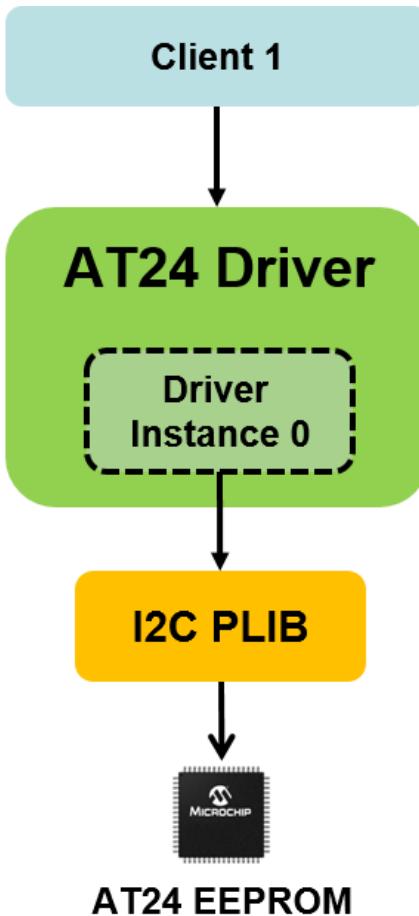
This library provides an abstraction of the AT24 Driver library.

Description

The AT24 library interface provides read and write functions that abstract out the internal workings of the AT24 driver and the underlying I2C (or TWIHS) protocol. The AT24 library supports a single instance of the AT24 EEPROM and a single client.

The client can be:

- Application - Directly access the AT24 EEPROM using the APIs provided by the AT24 library.
- Memory Driver - Application can run a file system on the AT24 EEPROM by connecting it to the Memory Driver which can further be connected to the File System Service.



How the Library Works

This topic describes the basic architecture of the AT24 Driver Library and provides information on how the library works.

Description

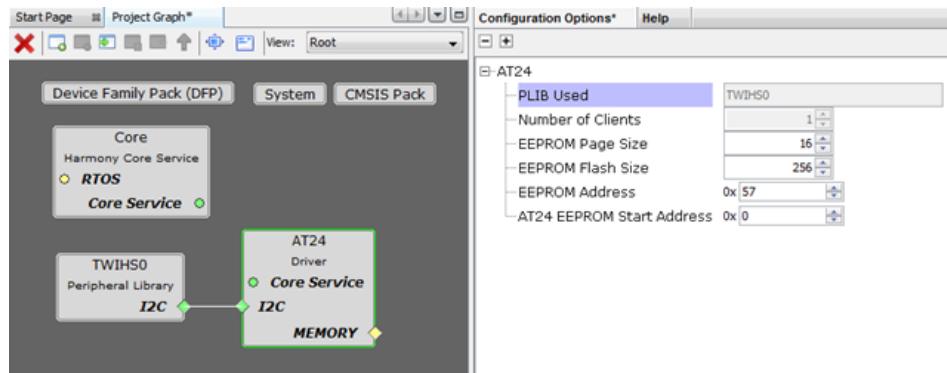
- The AT24 Driver Library registers an event handler with the underlying I2C (or TWIHS) peripheral library. This event handler is called by the PLIB from the interrupt context to notify the AT24 driver that the requested transfer is complete.
- The library's state machine is driven from the interrupt context. Once a transfer is complete a callback (if registered by the application) is given to the application from the interrupt context.
- The library does not support queuing of more than one requests. The application must check and ensure that any previous request is completed before submitting a new one. This can be done either by polling the status of the data transfer or registering a callback.

Configuring the Library

This Section provides information on how to configure the AT24 Driver library.

Description

AT24 Driver library should be configured via MHC. Below is the snapshot of the MHC configuration window for configuring the AT24 driver and a brief description of various configuration options.



User Configurations:

- PLIB Used**
 - Indicates the I2C (or TWIHS) peripheral instance used by the AT24 driver.
- Number of Clients**
 - Always set to 1 as it supports only a single client.
- EEPROM Page Size**
 - Size of one page of EEPROM memory (in bytes).
- EEPROM Flash Size**
 - Total size of the EEPROM memory (in bytes).
- EEPROM Address**
 - The 7-bit I2C slave address of the EEPROM.
- AT24 EEPROM Start Address**
 - The EEPROM memory start address. This is applicable when the AT24 driver is connected to the Memory block driver.

Building the Library

This section provides information on how the AT24 Driver Library can be built.

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Functions

	Name	Description
≡	DRV_AT24_Initialize	Initializes the AT24 EEPROM device
≡	DRV_AT24_Status	Gets the current status of the AT24 driver module.

b) Core Client Functions

	Name	Description
≡	DRV_AT24_Open	Opens the specified AT24 driver instance and returns a handle to it.

≡	DRV_AT24_Close	Closes the opened-instance of the AT24 driver.
≡	DRV_AT24_EventHandlerSet	Allows a client to identify a transfer event handling function for the driver to call back when the requested transfer has finished.

c) Data Transfer Functions

	Name	Description
≡	DRV_AT24_Write	Writes 'n' bytes of data starting at the specified address.
≡	DRV_AT24_PageWrite	Writes one page of data starting at the specified address.
≡	DRV_AT24_Read	Reads 'n' bytes of data from the specified start address of EEPROM.
≡	DRV_AT24_TransferStatusGet	Gets the current status of the transfer request.

d) Block Interface Functions

	Name	Description
≡	DRV_AT24_GeometryGet	Returns the geometry of the device.

e) Data Types and Constants

	Name	Description
	DRV_AT24_EVENT_HANDLER	Pointer to a AT24 Driver Event handler function
	DRV_AT24_GEOMETRY	Defines the data type for AT24 EEPROM Geometry details.
	DRV_AT24_TRANSFER_STATUS	Defines the data type for AT24 Driver transfer status.
	DRV_AT24_PLIB_CALLBACK_REGISTER	This is type DRV_AT24_PLIB_CALLBACK_REGISTER.
	DRV_AT24_PLIB_ERROR_GET	This is type DRV_AT24_PLIB_ERROR_GET.
	DRV_AT24_PLIB_IS_BUSY	This is type DRV_AT24_PLIB_IS_BUSY.
	DRV_AT24_I2C_ERROR	
	DRV_AT24_PLIB_READ	This is type DRV_AT24_PLIB_READ.
	DRV_AT24_INIT	Defines the data required to initialize the AT24 driver
	DRV_AT24_PLIB_WRITE	This is type DRV_AT24_PLIB_WRITE.
	DRV_AT24_PLIB_CALLBACK	This is type DRV_AT24_PLIB_CALLBACK.
	DRV_AT24_PLIB_WRITE_READ	This is type DRV_AT24_PLIB_WRITE_READ.
	DRV_AT24_PLIB_INTERFACE	Defines the data required to initialize the AT24 driver PLIB Interface.

Description

This section describes the API functions of the AT24 Driver library.

Refer to each section for a detailed description.

a) System Functions

DRV_AT24_Initialize Function

Initializes the AT24 EEPROM device

File

[drv_at24.h](#)

C

```
SYS_MODULE_OBJ DRV_AT24_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 AT24 EEPROM device driver making it ready for clients to open and use. The initialization data is specified by the init parameter. It is a single instance driver, so this API should be called only once.

Remarks

This routine must be called before any other DRV_AT24 routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Example

```

SYS_MODULE_OBJ sysObjDrvAT24;

DRV_AT24_PLIB_INTERFACE drvAT24PlibAPI = {
    .writeRead = (DRV_AT24_PLIB_WRITE_READ)TWIHS0_WriteRead,
    .write = (DRV_AT24_PLIB_WRITE)TWIHS0_Write,
    .read = (DRV_AT24_PLIB_READ)TWIHS0_Read,
    .isBusy = (DRV_AT24_PLIB_IS_BUSY)TWIHS0_IsBusy,
    .errorGet = (DRV_AT24_PLIB_ERROR_GET)TWIHS0_ErrorGet,
    .callbackRegister = (DRV_AT24_PLIB_CALLBACK_REGISTER)TWIHS0_CallbackRegister,
};

DRV_AT24_INIT drvAT24InitData = {
    .i2cPlib = &drvAT24PlibAPI,
    .slaveAddress = 0x57,
    .pageSize = DRV_AT24_EEPROM_PAGE_SIZE,
    .flashSize = DRV_AT24_EEPROM_FLASH_SIZE,
    .numClients = DRV_AT24_CLIENTS_NUMBER_IDX,
    .blockStartAddress = 0x0,
};

sysObjDrvAT24 = DRV_AT24_Initialize(DRV_AT24_INDEX, (SYS_MODULE_INIT *)&drvAT24InitData);

```

Parameters

Parameters	Description
drvIndex	Identifier for the instance to be initialized
init	Pointer to the init data structure containing any data necessary to initialize the driver.

Function

[SYS_MODULE_OBJ DRV_AT24_Initialize\(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init\)](#)

DRV_AT24_Status Function

Gets the current status of the AT24 driver module.

File

[drv_at24.h](#)

C

```
SYS_STATUS DRV_AT24_Status(const SYS_MODULE_INDEX drvIndex);
```

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 AT24 driver module.

Remarks

None.

Preconditions

Function [DRV_AT24_Initialize](#) should have been called before calling this function.

Example

```
SYS_STATUS status;

status = DRV_AT24_Status(DRV_AT24_INDEX);

if (status == SYS_STATUS_READY)
{
    // AT24 driver is initialized and ready to accept requests.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the instance used to initialize driver

Function

[SYS_STATUS DRV_AT24_Status\(const SYS_MODULE_INDEX drvIndex \)](#)

b) Core Client Functions

DRV_AT24_Open Function

Opens the specified AT24 driver instance and returns a handle to it.

File

[drv_at24.h](#)

C

```
DRV_HANDLE DRV_AT24_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 driver has been already opened once and in use.
- if the driver instance being opened is not initialized or is invalid.

Description

This routine opens the specified AT24 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.

This driver is a single client driver, so [DRV_AT24_Open](#) API should be called only once until driver is closed.

Remarks

This driver ignores the ioIntent argument.

The handle returned is valid until the [DRV_AT24_Close](#) routine is called.

Preconditions

Function `DRV_AT24_Initialize` must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_AT24_Open(DRV_AT24_INDEX, DRV_IO_INTENT_READWRITE);
if (handle == DRV_HANDLE_INVALID)
{
    // Unable to open the driver
    // May be the driver is not initialized
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration <code>DRV_IO_INTENT</code> "ORed" together to indicate the intended use of the driver.

Function

```
DRV_HANDLE DRV_AT24_Open
(
    const SYS_MODULE_INDEX drvIndex,
    const DRV_IO_INTENT ioIntent
)
```

DRV_AT24_Close Function

Closes the opened-instance of the AT24 driver.

File

`drv_at24.h`

C

```
void DRV_AT24_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This routine closes opened-instance of the AT24 driver, invalidating the handle. A new handle must be obtained by calling `DRV_AT24_Open` before the caller may use the driver again.

Remarks

None.

Preconditions

`DRV_AT24_Open` must have been called to obtain a valid opened device handle.

Example

```
// 'handle', returned from the DRV_AT24_Open
DRV_AT24_Close(handle);
```

Parameters

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

Function

```
void DRV_AT24_Close( const DRV_HANDLE handle )
```

DRV_AT24_EventHandlerSet Function

Allows a client to identify a transfer event handling function for the driver to call back when the requested transfer has finished.

File

[drv_at24.h](#)

C

```
void DRV_AT24_EventHandlerSet( const DRV_HANDLE handle, const DRV_AT24_EVENT_HANDLER
eventHandler, const uintptr_t context );
```

Returns

None.

Description

This function allows a client to register a transfer event handling function with the driver to call back when the requested transfer has finished.

The event handler should be set before the client submits any transfer requests 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

DRV_AT24_Open must have been called to obtain a valid opened device handle.

Example

```
#define BUFFER_SIZE 256
#define MEM_ADDRESS 0x0

// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[BUFFER_SIZE];

// myHandle is the handle returned from DRV_AT24_Open API.

// Client registers an event handler with driver. This is done once

DRV_AT24_EventHandlerSet( myHandle, APP_AT24TransferEventHandler, (uintptr_t)&myAppObj );

if (DRV_AT24_Read(myHandle, myBuffer, BUFFER_SIZE, MEM_ADDRESS) == false)
{
    // Error handling here
}

// The registered event handler is called when the request is complete.

void APP_AT24TransferEventHandler(DRV_AT24_TRANSFER_STATUS event, 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* pMyAppObj = (MY_APP_OBJ *) context;

switch(event)
{
    case DRV_AT24_TRANSFER_STATUS_COMPLETED:
        // This means the data was transferred.
        break;

    case DRV_AT24_TRANSFER_STATUS_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_AT24_EventHandlerSet(
    const DRV_HANDLE handle,
    const DRV_AT24_EVENT_HANDLER eventHandler,
    const uintptr_t context
)

```

c) Data Transfer Functions

DRV_AT24_Write Function

Writes 'n' bytes of data starting at the specified address.

File

[drv_at24.h](#)

C

```

bool DRV_AT24_Write(const DRV_HANDLE handle, void* txData, uint32_t txDataLength, uint32_t
address);

```

Returns

false

- if handle is not right
- if the pointer to the buffer to be written is NULL or number of bytes to write is zero
- if the driver is busy handling another transfer request

true

- if the write request is accepted.

Description

This function schedules a non-blocking write operation for writing txDataLength bytes of data starting from given address of EEPROM.

The requesting client should call [DRV_AT24_TransferStatusGet](#) API to know the current status of the request OR the requesting client can register a callback function with the driver to get notified of the status.

Remarks

None.

Preconditions

[DRV_AT24_Open](#) must have been called to obtain a valid opened device handle.

Example

```
#define BUFFER_SIZE 1024
#define MEM_ADDRESS 0x00

uint8_t writeBuffer[BUFFER_SIZE];

// myHandle is the handle returned from DRV_AT24_Open API.
// In the below example, the transfer status is polled. However, application can
// register a callback and get notified when the transfer is complete.

if (DRV_AT24_Write(myHandle, writeBuffer, BUFFER_SIZE, MEM_ADDRESS) != true)
{
    // Error handling here
}
else
{
    // Wait for write to be completed
    while(DRV_AT24_TransferStatusGet(myHandle) == DRV_AT24_TRANSFER_STATUS_BUSY);
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txData	The source buffer containing data to be programmed into AT24 EEPROM
txDataLength	Total number of bytes to be written.
address	Memory start address from where the data should be written

Function

bool DRV_AT24_Write(const [DRV_HANDLE](#) handle, void* txData, uint32_t txDataLength, uint32_t address)

DRV_AT24_PageWrite Function

Writes one page of data starting at the specified address.

File

[drv_at24.h](#)

C

```
bool DRV_AT24_PageWrite(const DRV_HANDLE handle, void * txData, uint32_t address);
```

Returns

false

- if handle is not right
- if the pointer to the transmit data buffer is NULL
- if the driver is busy handling another transfer request

true

- if the write request is accepted.

Description

This function schedules a non-blocking write operation for writing one page of data starting from given address of EEPROM.

The requesting client should call [DRV_AT24_TransferStatusGet](#) API to know the current status of the request OR the requesting client can register a callback function with the driver to get notified of the status.

Remarks

None.

Preconditions

[DRV_AT24_Open](#) must have been called to obtain a valid opened device handle.

"address" provided must be page boundary aligned in order to avoid overwriting the data in the beginning of the page.

Example

```
#define BUFFER_SIZE 1024
#define MEM_ADDRESS 0x00

uint8_t writeBuffer[BUFFER_SIZE];

// myHandle is the handle returned from DRV_AT24_Open API.
// In the below example, the transfer status is polled. However, application can
// register a callback and get notified when the transfer is complete.

if (DRV_AT24_PageWrite(myHandle, writeBuffer, MEM_ADDRESS) != true)
{
    // Error handling here
}
else
{
    // Wait for write to be completed
    while(DRV_AT24_TransferStatusGet(myHandle) == DRV_AT24_TRANSFER_STATUS_BUSY);
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txData	The source buffer containing data to be programmed into AT24 EEPROM
address	Memory start address from where the data should be written. It must be page boundary aligned in order to avoid overwriting the data in the beginning of the page.

Function

```
bool DRV_AT24_PageWrite(const DRV_HANDLE handle, void* txData, uint32_t address)
```

DRV_AT24_Read Function

Reads 'n' bytes of data from the specified start address of EEPROM.

File

[drv_at24.h](#)

C

```
bool DRV_AT24_Read(const DRV_HANDLE handle, void * rxData, uint32_t rxDataLength, uint32_t address);
```

Returns

false

- if handle is not right
 - if the receive buffer pointer is NULL or number of bytes to read is zero
 - if the driver is busy handling another transfer request
- true
- if the read request is accepted.

Description

This function schedules a non-blocking read operation for the requested number of data bytes from given address of EEPROM.

The requesting client should call [DRV_AT24_TransferStatusGet](#) API to know the current status of the request OR the requesting client can register a callback function with the driver to get notified of the status.

Remarks

None.

Preconditions

[DRV_AT24_Open](#) must have been called to obtain a valid opened device handle.

Example

```
#define BUFFER_SIZE 1024
#define MEM_ADDRESS 0x00

uint8_t readBuffer[BUFFER_SIZE];

// myHandle is the handle returned from DRV_AT24_Open API.

// In the below example, the transfer status is polled. However, application can
// register a callback and get notified when the transfer is complete.

if (DRV_AT24_Read(myHandle, readBuffer, BUFFER_SIZE, MEM_ADDRESS) != true)
{
    // Error handling here
}
else
{
    // Wait for read to be completed
    while(DRV_AT24_TransferStatusGet(myHandle) == DRV_AT24_TRANSFER_STATUS_BUSY);
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
rxData	Buffer pointer into which the data read from the DRV_AT24 Flash memory will be placed.
rxDataLength	Total number of bytes to be read.
address	Memory start address from where the data should be read.

Function

bool DRV_AT24_Read(const [DRV_HANDLE](#) handle, void *rxData, uint32_t rxDataLength, uint32_t address)

DRV_AT24_TransferStatusGet Function

Gets the current status of the transfer request.

File

[drv_at24.h](#)

C

```
DRV_AT24_TRANSFER_STATUS DRV\_AT24\_TransferStatusGet(const DRV\_HANDLE handle);
```

Returns

One of the status element from the enum [DRV_AT24_TRANSFER_STATUS](#).

Description

This routine gets the current status of the transfer request.

Remarks

None.

Preconditions

[DRV_AT24_PageWrite](#) or [DRV_AT24_Read](#) must have been called to obtain the status of transfer.

Example

```
// myHandle is the handle returned from DRV_AT24_Open API.

if (DRV_AT24_TransferStatusGet(myHandle) == DRV_AT24_TRANSFER_STATUS_COMPLETED)
{
    // Operation Done
}
```

Parameters

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

Function

[DRV_AT24_TRANSFER_STATUS](#) `DRV_AT24_TransferStatusGet(const DRV_HANDLE handle)`

d) Block Interface Functions

DRV_AT24_GeometryGet Function

Returns the geometry of the device.

File

[drv_at24.h](#)

C

```
bool DRV_AT24_GeometryGet(const DRV_HANDLE handle, DRV_AT24_GEOMETRY * geometry);
```

Returns

false

- if handle is invalid

true

- if able to get the geometry details of the flash

Description

This API gives the following geometrical details of the DRV_AT24 Flash:

- Number of Read/Write/Erase Blocks and their size in each region of the device

Remarks

None.

Preconditions

[DRV_AT24_Open](#) must have been called to obtain a valid opened device handle.

Example

```
DRV_AT24_GEOMETRY eepromGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

// myHandle is the handle returned from DRV_AT24_Open API.

DRV_AT24_GeometryGet(myHandle, &eepromGeometry);

readBlockSize = eepromGeometry.readBlockSize;
nReadBlocks = eepromGeometry.readNumBlocks;
nReadRegions = eepromGeometry.readNumRegions;

writeBlockSize = eepromGeometry.writeBlockSize;
eraseBlockSize = eepromGeometry.eraseBlockSize;

totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
geometry	Pointer to flash device geometry table instance

Function

```
bool DRV_AT24_GeometryGet(const DRV_HANDLE handle, DRV_AT24_GEOMETRY *geometry)
```

e) Data Types and Constants

DRV_AT24_EVENT_HANDLER Type

Pointer to a AT24 Driver Event handler function

File

[drv_at24.h](#)

C

```
typedef void (* DRV_AT24_EVENT_HANDLER)(DRV_AT24_TRANSFER_STATUS event, uintptr_t context);
```

Returns

None.

Description

AT24 Driver Transfer Event Handler Function Pointer

This data type defines the required function signature for the AT24 driver event handling callback function. A client must register a pointer using the event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive transfer 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_AT24_TRANSFER_STATUS_COMPLETED, it means that the data was transferred successfully.

If the event is DRV_AT24_TRANSFER_STATUS_ERROR, it means that the data was not transferred successfully.

The context parameter contains the handle to the client context, provided at the time the event handling function was registered using the [DRV_AT24_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 buffer add request.

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

The [DRV_AT24_Read](#), [DRV_AT24_Write](#) and [DRV_AT24_PageWrite](#) functions can be called in the event handler to submit a new request to the driver.

Example

```
void APP_MyTransferEventHandler( DRV_AT24_TRANSFER_STATUS event, uintptr_t context )
{
    MY_APP_DATA_STRUCT* pAppData = (MY_APP_DATA_STRUCT *) context;

    switch(event)
    {
        case DRV_AT24_TRANSFER_STATUS_COMPLETED:
            // Handle the transfer complete event.
            break;

        case DRV_AT24_TRANSFER_STATUS_ERROR:
        default:
            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
context	Value identifying the context of the application that registered the event handling function.

DRV_AT24_GEOMETRY Structure

Defines the data type for AT24 EEPROM Geometry details.

File

[drv_at24.h](#)

C

```
typedef struct {
    uint32_t readBlockSize;
    uint32_t readNumBlocks;
    uint32_t readNumRegions;
    uint32_t writeBlockSize;
    uint32_t writeNumBlocks;
    uint32_t writeNumRegions;
    uint32_t eraseBlockSize;
    uint32_t eraseNumBlocks;
    uint32_t eraseNumRegions;
    uint32_t blockStartAddress;
} DRV_AT24_GEOMETRY;
```

Description

DRV_AT24 Geometry data

This will be used to get the geometry details of the attached AT24 EEPROM device.

Remarks

None.

DRV_AT24_TRANSFER_STATUS Enumeration

Defines the data type for AT24 Driver transfer status.

File

[drv_at24.h](#)

C

```
typedef enum {
    DRV_AT24_TRANSFER_STATUS_BUSY,
    DRV_AT24_TRANSFER_STATUS_COMPLETED,
    DRV_AT24_TRANSFER_STATUS_ERROR
} DRV_AT24_TRANSFER_STATUS;
```

Members

Members	Description
DRV_AT24_TRANSFER_STATUS_BUSY	Transfer is being processed
DRV_AT24_TRANSFER_STATUS_COMPLETED	Transfer is successfully completed
DRV_AT24_TRANSFER_STATUS_ERROR	Transfer had error

Description

DRV_AT24 Transfer Status

This will be used to indicate the current transfer status of the AT24 EEPROM driver operations.

Remarks

None.

DRV_AT24_PLIB_CALLBACK_REGISTER Type

File

[drv_at24_definitions.h](#)

C

```
typedef void (* DRV_AT24_PLIB_CALLBACK_REGISTER)(DRV_AT24_PLIB_CALLBACK, uintptr_t);
```

Description

This is type DRV_AT24_PLIB_CALLBACK_REGISTER.

DRV_AT24_PLIB_ERROR_GET Type

File

[drv_at24_definitions.h](#)

C

```
typedef DRV_AT24_I2C_ERROR (* DRV_AT24_PLIB_ERROR_GET)(void);
```

Description

This is type DRV_AT24_PLIB_ERROR_GET.

DRV_AT24_PLIB_IS_BUSY Type**File**[drv_at24_definitions.h](#)**C**

```
typedef bool (* DRV_AT24_PLIB_IS_BUSY)(void);
```

Description

This is type DRV_AT24_PLIB_IS_BUSY.

DRV_AT24_I2C_ERROR Enumeration**File**[drv_at24_definitions.h](#)**C**

```
typedef enum {
    DRV_AT24_I2C_ERROR_NONE = 0,
    DRV_AT24_I2C_ERROR_NACK
} DRV_AT24_I2C_ERROR;
```

Members

Members	Description
DRV_AT24_I2C_ERROR_NACK	Slave returned Nack

Section

Data Types

DRV_AT24_PLIB_READ Type**File**[drv_at24_definitions.h](#)**C**

```
typedef bool (* DRV_AT24_PLIB_READ)(uint16_t , uint8_t* , uint32_t);
```

Description

This is type DRV_AT24_PLIB_READ.

DRV_AT24_INIT Structure

Defines the data required to initialize the AT24 driver

File[drv_at24_definitions.h](#)**C**

```
typedef struct {
    const DRV_AT24_PLIB_INTERFACE* i2cPlib;
    uint16_t slaveAddress;
```

```

    uint32_t pageSize;
    uint32_t flashSize;
    size_t numClients;
    uint32_t blockStartAddress;
} DRV_AT24_INIT;

```

Members

Members	Description
const DRV_AT24_PLIB_INTERFACE* i2cPlib;	Identifies the PLIB API set to be used by the driver to access the • peripheral.
uint16_t slaveAddress;	Address of the I2C slave
uint32_t pageSize;	Page size (in Bytes) of the EEPROM
uint32_t flashSize;	Total size (in Bytes) of the EEPROM
size_t numClients;	Number of clients

Description

AT24 Driver Initialization Data

This data type defines the data required to initialize or the AT24 driver.

Remarks

None.

DRV_AT24_PLIB_WRITE Type

File

[drv_at24_definitions.h](#)

C

```
typedef bool (* DRV_AT24_PLIB_WRITE)(uint16_t , uint8_t* , uint32_t);
```

Description

This is type DRV_AT24_PLIB_WRITE.

DRV_AT24_PLIB_CALLBACK Type

File

[drv_at24_definitions.h](#)

C

```
typedef void (* DRV_AT24_PLIB_CALLBACK)(uintptr_t);
```

Description

This is type DRV_AT24_PLIB_CALLBACK.

DRV_AT24_PLIB_WRITE_READ Type

File

[drv_at24_definitions.h](#)

C

```
typedef bool (* DRV_AT24_PLIB_WRITE_READ)(uint16_t , uint8_t* , uint32_t , uint8_t* , uint32_t);
```

Description

This is type DRV_AT24_PLIB_WRITE_READ.

DRV_AT24_PLIB_INTERFACE Structure

Defines the data required to initialize the AT24 driver PLIB Interface.

File

[drv_at24_definitions.h](#)

C

```
typedef struct {
    DRV_AT24_PLIB_WRITE_READ writeRead;
    DRV_AT24_PLIB_WRITE write;
    DRV_AT24_PLIB_READ read;
    DRV_AT24_PLIB_IS_BUSY isBusy;
    DRV_AT24_PLIB_ERROR_GET errorGet;
    DRV_AT24_PLIB_CALLBACK_REGISTER callbackRegister;
} DRV_AT24_PLIB_INTERFACE;
```

Members

Members	Description
DRV_AT24_PLIB_WRITE_READ writeRead;	AT24 PLIB writeRead API
DRV_AT24_PLIB_WRITE write;	AT24 PLIB write API
DRV_AT24_PLIB_READ read;	AT24 PLIB read API
DRV_AT24_PLIB_IS_BUSY isBusy;	AT24 PLIB Transfer status API
DRV_AT24_PLIB_ERROR_GET errorGet;	AT24 PLIB Error get API
DRV_AT24_PLIB_CALLBACK_REGISTER callbackRegister;	AT24 PLIB callback register API

Description

AT24 Driver PLIB Interface Data

This data type defines the data required to initialize the AT24 driver PLIB Interface.

Remarks

None.

Files

Files

Name	Description
drv_at24.h	AT24 EEPROM Library Interface header.
drv_at24_definitions.h	AT24 Driver Definitions Header File

Description

This section will list only the library's interface header file(s).

drv_at24.h

AT24 EEPROM Library Interface header.

Enumerations

	Name	Description
	DRV_AT24_TRANSFER_STATUS	Defines the data type for AT24 Driver transfer status.

Functions

	Name	Description
≡	DRV_AT24_Close	Closes the opened-instance of the AT24 driver.
≡	DRV_AT24_EventHandlerSet	Allows a client to identify a transfer event handling function for the driver to call back when the requested transfer has finished.
≡	DRV_AT24_GeometryGet	Returns the geometry of the device.
≡	DRV_AT24_Initialize	Initializes the AT24 EEPROM device
≡	DRV_AT24_Open	Opens the specified AT24 driver instance and returns a handle to it.
≡	DRV_AT24_PageWrite	Writes one page of data starting at the specified address.
≡	DRV_AT24_Read	Reads 'n' bytes of data from the specified start address of EEPROM.
≡	DRV_AT24_Status	Gets the current status of the AT24 driver module.
≡	DRV_AT24_TransferStatusGet	Gets the current status of the transfer request.
≡	DRV_AT24_Write	Writes 'n' bytes of data starting at the specified address.

Structures

	Name	Description
	DRV_AT24_GEOMETRY	Defines the data type for AT24 EEPROM Geometry details.

Types

	Name	Description
	DRV_AT24_EVENT_HANDLER	Pointer to a AT24 Driver Event handler function

Description

DRV_AT24 Driver Interface Definition

The AT24 Driver Library provides a interface to access the AT24 family of EEPROMs.

File Name

drv_at24.h

Company

Microchip Technology Inc.

drv_at24_definitions.h

AT24 Driver Definitions Header File

Enumerations

	Name	Description
	DRV_AT24_I2C_ERROR	

Structures

	Name	Description
	DRV_AT24_INIT	Defines the data required to initialize the AT24 driver
	DRV_AT24_PLIB_INTERFACE	Defines the data required to initialize the AT24 driver PLIB Interface.

Types

	Name	Description
	DRV_AT24_PLIB_CALLBACK	This is type DRV_AT24_PLIB_CALLBACK.

	DRV_AT24_PLIB_CALLBACK_REGISTER	This is type DRV_AT24_PLIB_CALLBACK_REGISTER.
	DRV_AT24_PLIB_ERROR_GET	This is type DRV_AT24_PLIB_ERROR_GET.
	DRV_AT24_PLIB_IS_BUSY	This is type DRV_AT24_PLIB_IS_BUSY.
	DRV_AT24_PLIB_READ	This is type DRV_AT24_PLIB_READ.
	DRV_AT24_PLIB_WRITE	This is type DRV_AT24_PLIB_WRITE.
	DRV_AT24_PLIB_WRITE_READ	This is type DRV_AT24_PLIB_WRITE_READ.

Description

AT24 Driver Definitions Header File

This file provides implementation-specific definitions for the AT24 driver's system interface.

File Name

drv_at24_definitions.h

Company

Microchip Technology Inc.

AT25 Driver Library Help

This section describes the AT25 External EEPROM Driver Library.

Introduction

This library provides an interface to access the external AT25 EEPROM over the SPI interface.

Description

This library provides a non-blocking interface to read and write to the external AT25 EEPROM. The library uses the SPI peripheral library (PLIB) to interface with the AT25 EEPROM.

Key Features:

- Supports a single instance of the AT25 EEPROM and a single client to the driver.
- Supports page writes.
- Supports writes to random memory address and across page boundaries.
- The library interface is compliant to the block media interface expected by the Memory Driver. This allows running a file system on the AT25 EEPROM using the Memory Driver and the File System Service.
- The library can be used in both bare-metal and RTOS environments.

Using the Library

This topic describes the basic architecture of the AT25 Library and provides information on how to use the library.

Description

The AT25 library provides non-blocking APIs to read and write to external AT25 EEPROM. It uses the SPI peripheral library to interface with the AT25 EEPROM.

- The library provides APIs to perform reads/writes from/to any EEPROM memory address, with number of bytes spanning multiple pages.
- The library provides API to perform page write to EEPROM. Here, the memory start address must be aligned to the EEPROM page boundary.
- Application can either register a callback to get notified once the data transfer is complete or can poll the status of the data transfer.
- The library interface complies to the block driver interface expected by the Memory Driver. This allows application to run a file system on the AT25 EEPROM media using the Memory Driver and the File System Service.
- The library can be used in both bare-metal and RTOS environments.

Abstraction Model

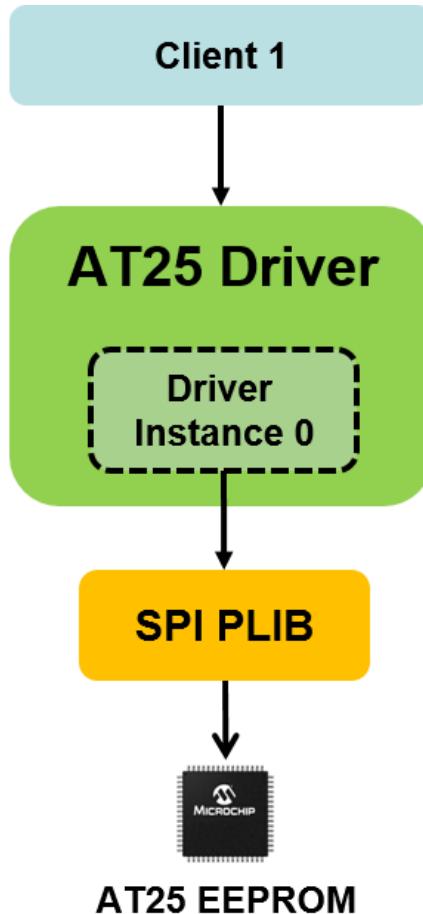
This library provides a low-level abstraction of the AT25 EEPROM Driver Library with a convenient C language interface.

Description

The AT25 library interface provides read and write functions that abstract out the internal workings of the AT25 driver and the underlying SPI protocol. The AT25 library supports a single instance of the AT25 EEPROM and a single client.

The client can be:

- Application - Directly access the AT25 EEPROM using the APIs provided by the AT25 library.
- Memory Driver - Application can run a file system on the AT25 EEPROM by connecting it to the Memory Driver which can further be connected to the File System Service.



How the Library Works

This topic provides information on how the AT25 Driver Library works.

Description

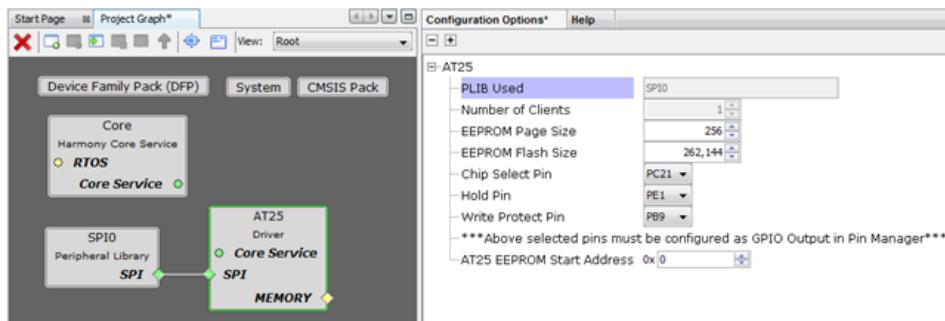
- The AT25 Driver Library registers an event handler with the underlying SPI peripheral library (PLIB). This event handler is called by the PLIB from the interrupt context to notify the AT25 driver that the requested transfer is complete.
- The library's state machine is driven from the interrupt context. Once a transfer is complete a callback (if registered by the application) is given to the application from the interrupt context.
- The library does not support queuing of more than one requests. The application must check and ensure that any previous request is completed before submitting a new one. This can be done either by polling the status of the data transfer or registering a callback.

Configuring the Library

This Section provides information on how to configure the AT25 Driver library.

Description

AT25 Driver library should be configured via MHC. Below is the snapshot of the MHC configuration window for configuring the AT25 driver and a brief description of various configuration options.



User Configurations:

- **PLIB Used**
 - Indicates the SPI peripheral instance used by the AT25 driver.
- **Number of Clients**
 - Always set to 1 as it supports only a single client.
- **EEPROM Page Size**
 - Size of one page of EEPROM memory (in bytes).
- **EEPROM Flash Size**
 - Total size of the EEPROM memory (in bytes). Depending on the specified EEPROM Flash Size, the driver will generate the appropriate number of address bits (8-bit, 16-bit or 24-bit), thereby allowing it to communicate with EEPROM of different sizes in the AT25 family.
- **Chip Select Pin**
 - EEPROM chip select pin (active low). This pin must be configured as GPIO output in "Pin Settings" configuration.
- **Hold Pin**
 - EEPROM hold pin (active low). This pin must be configured as GPIO output in "Pin Settings" configuration.
- **Write Protect Pin**
 - EEPROM write protect pin (active low). This pin must be configured as GPIO output in "Pin Settings" configuration. The AT25 driver keeps the Write Protect pin in logic high state, which means writes are always allowed.
- **AT25 EEPROM Start Address**
 - The EEPROM memory start address. This is applicable only when the AT25 driver is connected to the Memory Block Driver.

Building the Library

This section provides information on how the AT25 Driver Library can be built.

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Functions

	Name	Description
≡	DRV_AT25_Initialize	Initializes the AT25 EEPROM device
≡	DRV_AT25_Status	Gets the current status of the AT25 driver module.

b) Core Client Functions

	Name	Description
≡	DRV_AT25_Open	Opens the specified AT25 driver instance and returns a handle to it.
≡	DRV_AT25_Close	Closes the opened-instance of the AT25 driver.
≡	DRV_AT25_EventHandlerSet	Allows a client to identify a transfer event handling function for the driver to call back when the requested transfer has finished.

c) Data Transfer Functions

	Name	Description
≡	DRV_AT25_Read	Reads 'n' bytes of data from the specified start address of EEPROM.
≡	DRV_AT25_Write	Writes 'n' bytes of data starting at the specified address.
≡	DRV_AT25_PageWrite	Writes one page of data starting at the specified address.
≡	DRV_AT25_TransferStatusGet	Gets the current status of the transfer request.

d) Block Interface Functions

	Name	Description
≡	DRV_AT25_GeometryGet	Returns the geometry of the device.

e) Data Types and Constants

	Name	Description
	DRV_AT25_GEOMETRY	Defines the data type for AT25 EEPROM Geometry details.
	DRV_AT25_TRANSFER_STATUS	Defines the data type for AT25 Driver transfer status.
	DRV_AT25_EVENT_HANDLER	Pointer to a AT25 Driver Event handler function

Description

This section describes the API functions of the AT25 Driver library.

Refer to each section for a detailed description.

a) System Functions

DRV_AT25_Initialize Function

Initializes the AT25 EEPROM device

File

[drv_at25.h](#)

C

```
SYS_MODULE_OBJ DRV_AT25_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 AT25 EEPROM device driver making it ready for clients to open and use. The initialization data is specified by the init parameter. It is a single instance driver, so this API should be called only once.

Remarks

This routine must be called before any other DRV_AT25 routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Example

```
SYS_MODULE_OBJ sysObjDrvAT25;

DRV_AT25_PLIB_INTERFACE drvAT25PlibAPI = {
    .writeRead = (DRV_AT25_PLIB_WRITE_READ)SPI0_WriteRead,
    .write = (DRV_AT25_PLIB_WRITE)SPI0_Write,
    .read = (DRV_AT25_PLIB_READ)SPI0_Read,
    .isBusy = (DRV_AT25_PLIB_IS_BUSY)SPI0_IsBusy,
    .callbackRegister = (DRV_AT25_PLIB_CALLBACK_REGISTER)SPI0_CallbackRegister,
};

DRV_AT25_INIT drvAT25InitData = {
    .spiPlib = &drvAT25PlibAPI,
    .numClients = DRV_AT25_CLIENTS_NUMBER_IDX,
    .pageSize = DRV_AT25_EEPROM_PAGE_SIZE,
    .flashSize = DRV_AT25_EEPROM_FLASH_SIZE,
    .blockStartAddress = 0x0,
    .chipSelectPin = DRV_AT25_CHIP_SELECT_PIN_IDX,
    .holdPin = DRV_AT25_HOLD_PIN_IDX,
    .writeProtectPin = DRV_AT25_WP_PIN_IDX,
};

sysObjDrvAT25 = DRV_AT25_Initialize(DRV_AT25_INDEX, (SYS_MODULE_INIT *)&drvAT25InitData);
```

Parameters

Parameters	Description
drvIndex	Identifier for the instance to be initialized.
init	Pointer to the init data structure containing any data necessary to initialize the driver.

Function

```
SYS_MODULE_OBJ DRV_AT25_Initialize(
    const SYS_MODULE_INDEX drvIndex,
    const SYS_MODULE_INIT * const init
)
```

DRV_AT25_Status Function

Gets the current status of the AT25 driver module.

File

[drv_at25.h](#)

C

```
SYS_STATUS DRV_AT25_Status(const SYS_MODULE_INDEX drvIndex);
```

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 AT25 driver module.

Remarks

None.

Preconditions

Function [DRV_AT25_Initialize](#) should have been called before calling this function.

Example

```
SYS_STATUS status;  
  
status = DRV_AT25_Status(DRV_AT25_INDEX);
```

Parameters

Parameters	Description
drvIndex	Identifier for the instance used to initialize driver

Function

[SYS_STATUS DRV_AT25_Status\(const SYS_MODULE_INDEX drvIndex \)](#)

b) Core Client Functions**DRV_AT25_Open Function**

Opens the specified AT25 driver instance and returns a handle to it.

File

[drv_at25.h](#)

C

```
DRV_HANDLE DRV_AT25_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 driver has been already opened once and in use.
- if the driver instance being opened is not initialized or is invalid.

Description

This routine opens the specified AT25 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.

This driver is a single client driver, so DRV_AT25_Open API should be called only once until driver is closed.

Remarks

This driver ignores the ioIntent argument.

The handle returned is valid until the [DRV_AT25_Close](#) routine is called.

Preconditions

Function [DRV_AT25_Initialize](#) must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_AT25_Open(DRV_AT25_INDEX, DRV_IO_INTENT_READWRITE);
if (handle == DRV_HANDLE_INVALID)
{
    // Unable to open the driver
    // May be the driver is not initialized
}
```

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_AT25_Open
(
    const SYS_MODULE_INDEX drvIndex,
    const DRV_IO_INTENT ioIntent
)
```

DRV_AT25_Close Function

Closes the opened-instance of the AT25 driver.

File

[drv_at25.h](#)

C

```
void DRV_AT25_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This routine closes opened-instance of the AT25 driver, invalidating the handle. A new handle must be obtained by calling [DRV_AT25_Open](#) before the caller may use the driver again.

Remarks

None.

Preconditions

[DRV_AT25_Open](#) must have been called to obtain a valid opened device handle.

Example

```
// 'handle', returned from the DRV_AT25_Open
```

```
DRV_AT25_Close(handle);
```

Parameters

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

Function

```
void DRV_AT25_Close( const DRV_Handle handle )
```

DRV_AT25_EventHandlerSet Function

Allows a client to identify a transfer event handling function for the driver to call back when the requested transfer has finished.

File

[drv_at25.h](#)

C

```
void DRV_AT25_EventHandlerSet( const DRV_HANDLE handle, const DRV_AT25_EVENT_HANDLER
eventHandler, const uintptr_t context );
```

Returns

None.

Description

This function allows a client to register a transfer event handling function with the driver to call back when the requested transfer has finished.

The event handler should be set before the client submits any transfer requests 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

[DRV_AT25_Open](#) must have been called to obtain a valid opened device handle.

Example

```
#define BUFFER_SIZE 256
#define MEM_ADDRESS 0x00

// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[BUFFER_SIZE];

// myHandle is the handle returned from DRV_AT25_Open API.

// Client registers an event handler with driver. This is done once

DRV_AT25_EventHandlerSet( myHandle, APP_AT25TransferEventHandler, (uintptr_t)&myAppObj );

if (DRV_AT25_Read(myHandle, myBuffer, BUFFER_SIZE, MEM_ADDRESS) == false)
{
    // Error handling here
}

// The registered event handler is called when the request is complete.

void APP_AT25TransferEventHandler(DRV_AT25_TRANSFER_STATUS event, 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* pMyAppObj = (MY_APP_OBJ *) context;

switch(event)
{
    case DRV_AT25_TRANSFER_STATUS_COMPLETED:
        // This means the data was transferred.
        break;

    case DRV_AT25_TRANSFER_STATUS_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_AT25_EventHandlerSet(
    const DRV_HANDLE handle,
    const DRV_AT25_EVENT_HANDLER eventHandler,
    const uintptr_t context
)

```

c) Data Transfer Functions

DRV_AT25_Read Function

Reads 'n' bytes of data from the specified start address of EEPROM.

File

[drv_at25.h](#)

C

```

bool DRV_AT25_Read(const DRV_HANDLE handle, void * rxData, uint32_t rxDataLength, uint32_t
address);

```

Returns

false

- if handle is invalid
- if the pointer to the receive buffer is NULL or number of bytes to read is 0
- if the driver is busy handling another transfer request

true

- if the read request is accepted.

Description

This function schedules a non-blocking read operation for the requested number of data bytes from the given address of the EEPROM.

The requesting client should call [DRV_AT25_TransferStatusGet](#) API to know the current status of the request OR the requesting client can register a callback function with the driver to get notified of the status.

Remarks

None.

Preconditions

[DRV_AT25_Open](#) must have been called to obtain a valid opened device handle.

Example

```
#define BUFFER_SIZE 1024
#define MEM_ADDRESS 0x00

uint8_t readBuffer[BUFFER_SIZE];

// myHandle is the handle returned from DRV_AT25_Open API.
// In the below example, the transfer status is polled. However, application can
// register a callback and get notified when the transfer is complete.

if (DRV_AT25_Read(myHandle, readBuffer, BUFFER_SIZE, MEM_ADDRESS) != true)
{
    // Error handling here
}
else
{
    // Wait for read to be completed
    while(DRV_AT25_TransferStatusGet(myHandle) == DRV_AT25_TRANSFER_STATUS_BUSY);
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
rxData	Buffer pointer into which the data read from the DRV_AT25 Flash memory will be placed.
rxDataLength	Total number of bytes to be read.
address	Memory start address from where the data should be read.

Function

bool DRV_AT25_Read(const [DRV_HANDLE](#) handle, void *rxData, uint32_t rxDataLength, uint32_t address)

DRV_AT25_Write Function

Writes 'n' bytes of data starting at the specified address.

File

[drv_at25.h](#)

C

```
bool DRV_AT25_Write(const DRV_HANDLE handle, void * txData, uint32_t txDataLength, uint32_t
address);
```

Returns

false

- if handle is invalid

- if the pointer to transmit buffer is NULL or number of bytes to write is 0
 - if the driver is busy handling another transfer request
- true
- if the write request is accepted.

Description

This function schedules a non-blocking write operation for writing txDataLength bytes of data starting from given address of EEPROM.

The requesting client should call [DRV_AT25_TransferStatusGet](#) API to know the current status of the request OR the requesting client can register a callback function with the driver to get notified of the status.

Remarks

None.

Preconditions

[DRV_AT25_Open](#) must have been called to obtain a valid opened device handle.

Example

```

#define PAGE_SIZE      256
#define BUFFER_SIZE    1024
#define MEM_ADDRESS    0x00

uint8_t writeBuffer[BUFFER_SIZE];

// myHandle is the handle returned from DRV_AT25_Open API.
// In the below example, the transfer status is polled. However, application can
// register a callback and get notified when the transfer is complete.

if (DRV_AT25_Write(myHandle, writeBuffer, BUFFER_SIZE, MEM_ADDRESS) != true)
{
    // Error handling here
}
else
{
    // Wait for write to be completed
    while(DRV_AT25_TransferStatusGet(myHandle) == DRV_AT25_TRANSFER_STATUS_BUSY);
}

```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txData	The source buffer containing data to be programmed into AT25 EEPROM
txDataLength	Total number of bytes to be written.
address	Memory start address from where the data should be written

Function

bool DRV_AT25_Write(const [DRV_HANDLE](#) handle, void *txData, uint32_t txDataLength, uint32_t address)

DRV_AT25_PageWrite Function

Writes one page of data starting at the specified address.

File

[drv_at25.h](#)

C

```
bool DRV_AT25_PageWrite(const DRV_HANDLE handle, void * txData, uint32_t address);
```

Returns

- false
- if handle is invalid
 - if the pointer to the transmit data is NULL
 - if the driver is busy handling another transfer request
- true
- if the write request is accepted.

Description

This function schedules a non-blocking write operation for writing one page of data starting from the given address of the EEPROM.

The requesting client should call [DRV_AT25_TransferStatusGet](#) API to know the current status of the request OR the requesting client can register a callback function with the driver to get notified of the status.

Remarks

None.

Preconditions

[DRV_AT25_Open](#) must have been called to obtain a valid opened device handle.

"address" provided must be page boundary aligned in order to avoid overwriting the data in the beginning of the page.

Example

```
#define PAGE_SIZE 256
#define MEM_ADDRESS 0x0

uint8_t writeBuffer[PAGE_SIZE];

// myHandle is the handle returned from DRV_AT25_Open API.
// In the below example, the transfer status is polled. However, application can
// register a callback and get notified when the transfer is complete.

if (DRV_AT25_PageWrite(myHandle, writeBuffer, MEM_ADDRESS) != true)
{
    // Error handling here
}
else
{
    // Wait for write to be completed
    while(DRV_AT25_TransferStatusGet(myHandle) == DRV_AT25_TRANSFER_STATUS_BUSY);
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txData	The source buffer containing data to be written to the AT25 EEPROM
address	Write memory start address from where the data should be written. It must be page boundary aligned in order to avoid overwriting the data in the beginning of the page.

Function

bool DRV_AT25_PageWrite(const [DRV_HANDLE](#) handle, void *txData, uint32_t address)

DRV_AT25_TransferStatusGet Function

Gets the current status of the transfer request.

File

[drv_at25.h](#)

C

```
DRV_AT25_TRANSFER_STATUS DRV_AT25_TransferStatusGet(const DRV_HANDLE handle);
```

Returns

One of the status element from the enum [DRV_AT25_TRANSFER_STATUS](#).

Description

This routine gets the current status of the transfer request.

Remarks

None.

Preconditions

[DRV_AT25_PageWrite](#), [DRV_AT25_Write](#) or [DRV_AT25_Read](#) must have been called to obtain the status of transfer.

Example

```
// myHandle is the handle returned from DRV_AT25_Open API.

if (DRV_AT25_TransferStatusGet(myHandle) == DRV_AT25_TRANSFER_STATUS_COMPLETED)
{
    // Operation Done
}
```

Parameters

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

Function

[DRV_AT25_TRANSFER_STATUS](#) [DRV_AT25_TransferStatusGet](#)(**const** [DRV_HANDLE](#) handle)

d) Block Interface Functions**DRV_AT25_GeometryGet Function**

Returns the geometry of the device.

File

[drv_at25.h](#)

C

```
bool DRV_AT25_GeometryGet(const DRV_HANDLE handle, DRV_AT25_GEOMETRY * geometry);
```

Returns

false

- if handle is invalid

true

- if able to get the geometry details of the flash

Description

This API gives the following geometrical details of the DRV_AT25 Flash:

- Number of Read/Write/Erase Blocks and their size in each region of the device

Remarks

None.

Preconditions

`DRV_AT25_Open` must have been called to obtain a valid opened device handle.

Example

```
DRV_AT25_GEOMETRY eepromGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

// myHandle is the handle returned from DRV_AT25_Open API.

DRV_AT25_GeometryGet(myHandle, &eepromGeometry);

readBlockSize = eepromGeometry.readBlockSize;
nReadBlocks = eepromGeometry.readNumBlocks;
nReadRegions = eepromGeometry.readNumRegions;

writeBlockSize = eepromGeometry.writeBlockSize;
eraseBlockSize = eepromGeometry.eraseBlockSize;

totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
geometry	Pointer to flash device geometry table instance

Function

bool `DRV_AT25_GeometryGet`(const `DRV_HANDLE` handle, `DRV_AT25_GEOMETRY` *geometry)

e) Data Types and Constants

DRV_AT25_GEOMETRY Structure

Defines the data type for AT25 EEPROM Geometry details.

File

`drv_at25.h`

C

```
typedef struct {
    uint32_t readBlockSize;
    uint32_t readNumBlocks;
    uint32_t readNumRegions;
    uint32_t writeBlockSize;
    uint32_t writeNumBlocks;
    uint32_t writeNumRegions;
    uint32_t eraseBlockSize;
    uint32_t eraseNumBlocks;
    uint32_t eraseNumRegions;
    uint32_t blockStartAddress;
} DRV_AT25_GEOMETRY;
```

Description

`DRV_AT25` Geometry data

This will be used to get the geometry details of the attached AT25 EEPROM device.

Remarks

None.

DRV_AT25_TRANSFER_STATUS Enumeration

Defines the data type for AT25 Driver transfer status.

File

[drv_at25.h](#)

C

```
typedef enum {
    DRV_AT25_TRANSFER_STATUS_BUSY,
    DRV_AT25_TRANSFER_STATUS_COMPLETED,
    DRV_AT25_TRANSFER_STATUS_ERROR
} DRV_AT25_TRANSFER_STATUS;
```

Members

Members	Description
DRV_AT25_TRANSFER_STATUS_BUSY	Transfer is being processed
DRV_AT25_TRANSFER_STATUS_COMPLETED	Transfer is successfully completed
DRV_AT25_TRANSFER_STATUS_ERROR	Transfer had error

Description

DRV_AT25 Transfer Status

This will be used to indicate the current transfer status of the AT25 EEPROM driver operations.

Remarks

None.

DRV_AT25_EVENT_HANDLER Type

Pointer to a AT25 Driver Event handler function

File

[drv_at25.h](#)

C

```
typedef void (* DRV_AT25_EVENT_HANDLER)(DRV_AT25_TRANSFER_STATUS event, uintptr_t context);
```

Returns

None.

Description

AT25 Driver Transfer Event Handler Function Pointer

This data type defines the required function signature for the AT25 driver event handling callback function. A client must register a pointer using the event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive transfer 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_AT25_TRANSFER_STATUS_COMPLETED, it means that the data was transferred successfully.

If the event is DRV_AT25_TRANSFER_STATUS_ERROR, it means that the data was not transferred successfully.

The context parameter contains the handle to the client context, provided at the time the event handling function was registered

using the [DRV_AT25_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 buffer add request.

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

The [DRV_AT25_Read](#), [DRV_AT25_Write](#) and [DRV_AT25_PageWrite](#) functions can be called in the event handler to submit a request to the driver.

Example

```
void APP_MyTransferEventHandler( DRV_AT25_TRANSFER_STATUS event, uintptr_t context )
{
    MY_APP_DATA_STRUCT* pAppData = (MY_APP_DATA_STRUCT*) context;

    switch(event)
    {
        case DRV_AT25_TRANSFER_STATUS_COMPLETED:
            // Handle the transfer complete event.
            break;

        case DRV_AT25_TRANSFER_STATUS_ERROR:
        default:
            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
context	Value identifying the context of the application that registered the event handling function.

Files

Files

Name	Description
drv_at25.h	AT25 EEPROM Library Interface header.

Description

This section will list only the library's interface header file(s).

[drv_at25.h](#)

AT25 EEPROM Library Interface header.

Enumerations

	Name	Description
	DRV_AT25_TRANSFER_STATUS	Defines the data type for AT25 Driver transfer status.

Functions

	Name	Description
	DRV_AT25_Close	Closes the opened-instance of the AT25 driver.

≡	DRV_AT25_EventHandlerSet	Allows a client to identify a transfer event handling function for the driver to call back when the requested transfer has finished.
≡	DRV_AT25_GeometryGet	Returns the geometry of the device.
≡	DRV_AT25_Initialize	Initializes the AT25 EEPROM device
≡	DRV_AT25_Open	Opens the specified AT25 driver instance and returns a handle to it.
≡	DRV_AT25_PageWrite	Writes one page of data starting at the specified address.
≡	DRV_AT25_Read	Reads 'n' bytes of data from the specified start address of EEPROM.
≡	DRV_AT25_Status	Gets the current status of the AT25 driver module.
≡	DRV_AT25_TransferStatusGet	Gets the current status of the transfer request.
≡	DRV_AT25_Write	Writes 'n' bytes of data starting at the specified address.

Structures

	Name	Description
	DRV_AT25_GEOMETRY	Defines the data type for AT25 EEPROM Geometry details.

Types

	Name	Description
	DRV_AT25_EVENT_HANDLER	Pointer to a AT25 Driver Event handler function

Description

DRV_AT25 Driver Interface Definition

The AT25 Driver Library provides a interface to access the AT25 external EEPROM.

File Name

drv_at25.h

Company

Microchip Technology Inc.

I2C Driver Library Help

This section describes the I2C Driver Library.

Introduction

This library provides an interface to manage the data transfer operations using the I2C module.

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 write, read and write-read transfers
- Support multi-client and multi-instance operation
- Provides data transfer events
- Supports blocking and non-blocking operation

Using the Library

This topic describes the basic architecture of the I2C Driver Library and provides information on how to use it.

Description

The I2C driver builds on top of the I2C (or TWIHS) peripheral library (PLIB) and provides write, read and write-read APIs in blocking and non-blocking mode.

- Provides Write, Read and Write followed by Read APIs.

- Supports multiple slaves connected to the same I2C peripheral instance (multi-client mode).
- In *asynchronous* (non-blocking) mode, application can either register a callback to get notified once the data transfer is complete or can poll the status of the data transfer using the status related APIs.
- In *asynchronous* mode, application can queue more than one requests without waiting for the previous request to be completed. The number of requests that can be queued depends on the depth of the transfer queue configured using MHC.
- The *asynchronous* mode is supported in both bare-metal and RTOS environment.
- The *synchronous* (blocking) mode of the driver provides a blocking behavior and is supported only in an RTOS environment.
- The *synchronous* mode of the driver does not support callback or queuing multiple requests. This is because the implementation is blocking in nature.

Abstraction Model

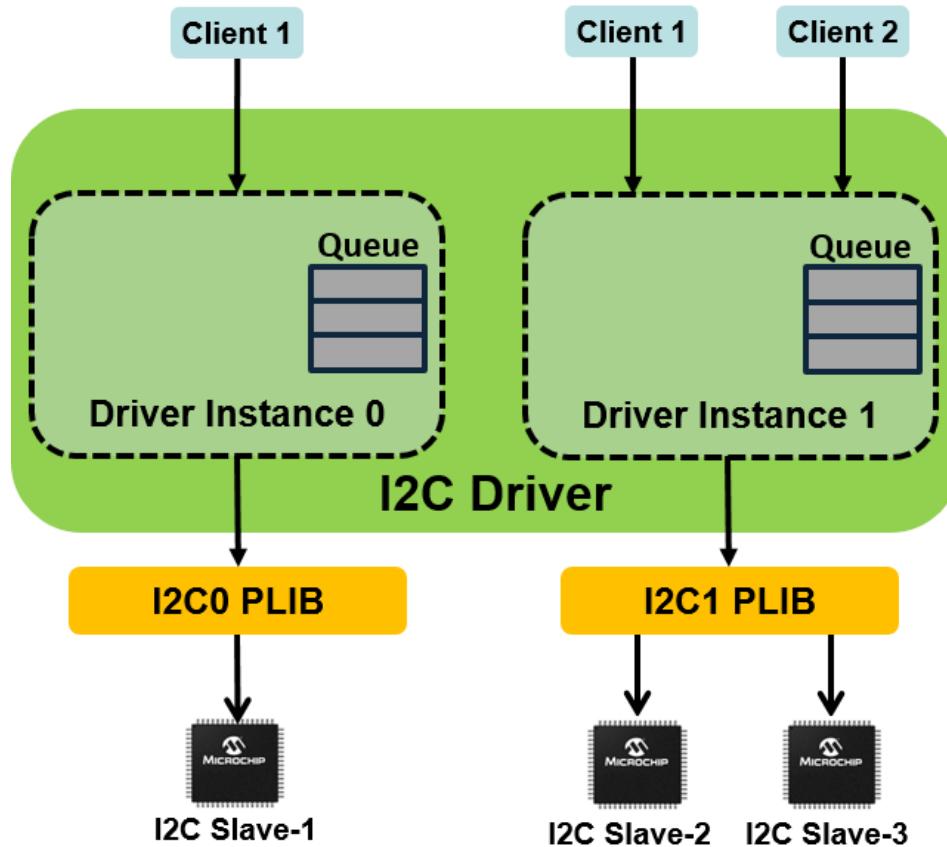
The I2C Driver Library provides the low-level abstraction of the I2C module with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the I2C Driver Library interface.

Description

The Driver initialization routines allow the application to initialize the driver. The initialization data configures the I2C module as a Master and sets the necessary parameters required for operation in the Master mode. The driver must be initialized before it can be used by the application.

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.



How the Library Works

This topic provides information on how the I2C Driver Library works.

Description

- The I2C driver is built on top of the I2C (or TWIHS) peripheral library.
- The I2C driver registers a callback with the underlying I2C peripheral library to receive transfer related events from the peripheral library. The I2C driver callback is called by the peripheral library from the interrupt context.
- The I2C driver state machine runs from the interrupt context. Once the transfer is complete, the driver calls the callback registered by the application (from the interrupt context).
- Each instance of the driver (in asynchronous/non-blocking mode) has a dedicated queue which can be configured using the MHC configuration options. The requests submitted by the clients are queued in the respective driver instance request queue.
- The I2C driver is capable of supporting multiple instances of the I2C peripheral.

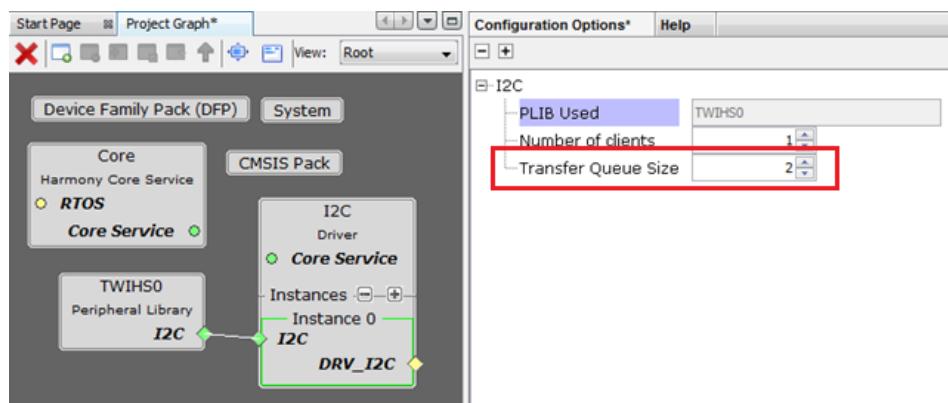
Configuring the Library

This Section provides information on how to configure the I2C Driver library.

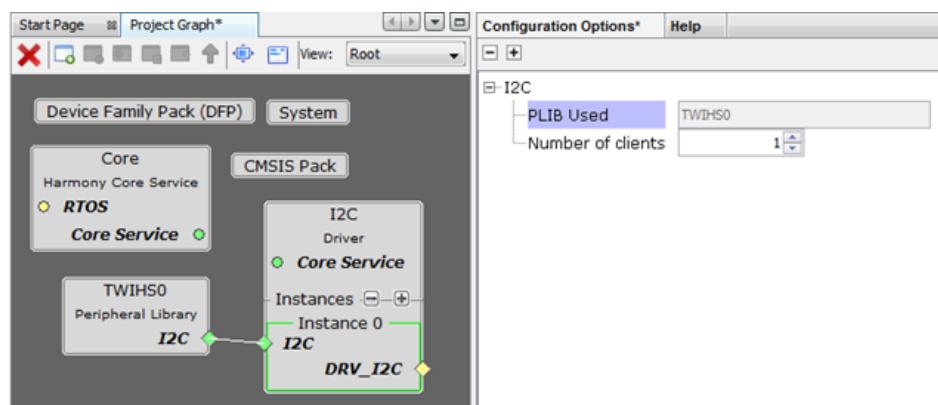
Description

I2C Driver library should be configured via MHC. Below is the snapshot of the MHC configuration window for configuring the I2C driver in asynchronous and synchronous mode and a brief description of various configuration options.

Asynchronous Mode



Synchronous Mode



Common User Configuration for all Instances

1. Driver Mode

1. Allows User to select the mode of driver(**Asynchronous or Synchronous**). This setting is common for all the instances

Instance Specific User Configurations

1. PLIB Used

1. Indicates the underlying I2C PLIB used by the driver.

2. Number of clients

1. The total number of clients that can open the given I2C driver instance.

3. Transfer Queue Size

1. Indicates the size of the transfer queue for the given I2C driver instance. **Available only in Asynchronous mode**

Building the Library

This section provides information on how the I2C Driver Library can be built.

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Functions

	Name	Description
≡	DRV_I2C_Initialize	Initializes the I2C instance for the specified driver index.
≡	DRV_I2C_Status	Gets the current status of the I2C driver module.

b) Core Client Functions

	Name	Description
≡	DRV_I2C_Open	Opens the specified I2C driver instance and returns a handle to it.
≡	DRV_I2C_Close	Closes an opened-instance of the I2C driver.
≡	DRV_I2C_TransferEventHandlerSet	Allows a client to identify a transfer event handling function for the driver to call back when queued transfers have finished.

c) Data Transfer Functions

	Name	Description
≡	DRV_I2C_ReadTransfer	This is a blocking function that performs a I2C read operation.
≡	DRV_I2C_ReadTransferAdd	Queues a read operation.
≡	DRV_I2C_WriteTransfer	This is a blocking function that performs a I2C write operation.
≡	DRV_I2C_WriteTransferAdd	Queues a write operation.
≡	DRV_I2C_WriteReadTransfer	This is a blocking function that performs a I2C write followed by a I2C read operation.
≡	DRV_I2C_WriteReadTransferAdd	Queues a write followed by read operation.
≡	DRV_I2C_TransferStatusGet	Returns the status of the write/read/write-read transfer request.
≡	DRV_I2C_ErrorGet	Gets the I2C hardware errors associated with the client.

d) Data Types and Constants

	Name	Description
	DRV_I2C_TRANSFER_HANDLE	Handle identifying a read, write or write followed by read transfer passed to the driver.
	DRV_I2C_TRANSFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_I2C_TRANSFER_EVENT_HANDLER	Pointer to a I2C Driver Transfer Event handler function

	DRV_I2C_TRANSFER_HANDLE_INVALID	Definition of an invalid transfer handle.
--	---	---

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 Functions

DRV_I2C_Initialize Function

Initializes the I2C instance for the specified driver index.

File

[drv_i2c.h](#)

C

```
SYS_MODULE_OBJ DRV_I2C_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 I2C 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 I2C peripheral instance it is associated with. For example, driver instance 0 can be assigned to I2C peripheral instance 2.

Remarks

This routine must be called before any other I2C routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Example

```
// The following code snippet shows an example I2C driver initialization.

SYS_MODULE_OBJ objectHandle;

DRV_I2C_CLIENT_OBJ drvI2C0ClientObjPool[DRV_I2C_CLIENTS_NUMBER_IDX0] = {0};
DRV_I2C_TRANSFER_OBJ drvI2C0TransferObjPool[DRV_I2C_QUEUE_SIZE_IDX0] = {0};

DRV_I2C_PLIB_INTERFACE drvI2C0PLibAPI = {
    .read = (DRV_I2C_PLIB_READ)TWIHS0_Read,
    .write = (DRV_I2C_PLIB_WRITE)TWIHS0_Write,
    .writeRead = (DRV_I2C_PLIB_WRITE_READ)TWIHS0_WriteRead,
    .errorGet = (DRV_I2C_PLIB_ERROR_GET)TWIHS0_ErrorGet,
    .callbackRegister = (DRV_I2C_PLIB_CALLBACK_REGISTER)TWIHS0_CallbackRegister,
};

DRV_I2C_INIT drvI2C0InitData = {

    .i2cPlib = &drvI2C0PLibAPI,
    .numClients = DRV_I2C_CLIENTS_NUMBER_IDX0
    .clientObjPool = (uintptr_t)&drvI2C0ClientObjPool[0],
    .interruptI2C = DRV_I2C_INT_SRC_IDX0,
```

```

    .queueSize = DRV_I2C_QUEUE_SIZE_IDX0,
    .transferObj = (uintptr_t)&drvI2C0TransferObj[0],
    .clockSpeed = DRV_I2C_CLOCK_SPEED_IDX0,
};

objectHandle = DRV_I2C_Initialize(DRV_I2C_INDEX_0, (SYS_MODULE_INIT*)&drvI2C0InitData);
if (objectHandle == SYS_MODULE_OBJ_INVALID)
{
    // Handle error
}

```

Parameters

Parameters	Description
drvIndex	Identifier for the instance to be initialized
init	Pointer to the init data structure containing any data necessary to initialize the driver.

Function

```

SYS_MODULE_OBJ DRV_I2C_Initialize
(
    const SYS_MODULE_INDEX drvIndex,
    const SYS_MODULE_INIT * const init
)

```

DRV_I2C_Status Function

Gets the current status of the I2C driver module.

File

[drv_i2c.h](#)

C

```
SYS_STATUS DRV_I2C_Status(const SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Initialization has succeeded and the I2C is ready for additional operations.

SYS_STATUS_UNINITIALIZED - Indicates that the driver is not initialized.

Description

This routine provides the current status of the I2C driver module.

Remarks

A driver can be opened only when its status is SYS_STATUS_READY.

Preconditions

Function [DRV_I2C_Initialize](#) should have been called before calling this function.

Example

```

SYS_MODULE_OBJ          object;      // Returned from DRV_I2C_Initialize
SYS_STATUS              i2cStatus;

i2cStatus = DRV_I2C_Status(object);
if (i2cStatus == SYS_STATUS_READY)
{
    // This means the driver can be opened using the
    // DRV_I2C_Open() function.
}

```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_I2C_Initialize routine

Function

[SYS_STATUS](#) [DRV_I2C_Status](#)(const [SYS_MODULE_OBJ](#) object)

b) Core Client Functions

DRV_I2C_Open Function

Opens the specified I2C driver instance and returns a handle to it.

File

[drv_i2c.h](#)

C

```
DRV_HANDLE DRV\_I2C\_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_I2C_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 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 I2C 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.

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_I2C_Close](#) routine is called.

Preconditions

Function [DRV_I2C_Initialize](#) must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV\_I2C\_Open(DRV_I2C_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (handle == DRV_HANDLE_INVALID)
{
    // 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_I2C_Open
(
const  SYS_MODULE_INDEX drvIndex,
const  DRV_IO_INTENT ioIntent
)
```

DRV_I2C_Close Function

Closes an opened-instance of the I2C driver.

File

[drv_i2c.h](#)

C

```
void DRV_I2C_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This routine closes an opened-instance of the I2C driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. A new handle must be obtained by calling [DRV_I2C_Open](#) before the caller may use the driver again.

Remarks

None.

Preconditions

[DRV_I2C_Open](#) must have been called to obtain a valid opened device handle.

Example

```
// 'handle', returned from the DRV_I2C_Open
DRV_I2C_Close(handle);
```

Parameters

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

Function

```
void DRV_I2C_Close( const DRV_HANDLE handle )
```

DRV_I2C_TransferEventHandlerSet Function

Allows a client to identify a transfer event handling function for the driver to call back when queued transfers have finished.

File

[drv_i2c.h](#)

C

```
void DRV_I2C_TransferEventHandlerSet(const DRV_HANDLE handle, const
DRV_I2C_TRANSFER_EVENT_HANDLER eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to register a transfer event handling function with the driver to call back when queued transfers have finished. When a client calls either the [DRV_I2C_ReadTransferAdd](#), [DRV_I2C_WriteTransferAdd](#) or [DRV_I2C_WriteReadTransferAdd](#) function, it is provided with a handle identifying the transfer that was added to the driver's transfer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the transfer has completed.

The event handler should be set before the client performs any "transfer 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. This function is available only in the asynchronous mode.

Preconditions

[DRV_I2C_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_I2C_TRANSFER_HANDLE transferHandle;

// myI2CHandle is the handle returned
// by the DRV_I2C_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2C_TransferEventHandlerSet( myI2CHandle, APP_I2CTransferEventHandler, (uintptr_t)&myAppObj
);

DRV_I2C_ReadTransferAdd(myI2CHandle, slaveAddress, myBuffer, MY_BUFFER_SIZE, &transferHandle);

if(transferHandle == DRV_I2C_TRANSFER_HANDLE_INVALID)
{
    // Error handling here
}

// The registered event handler is called when the transfer is completed.

void APP_I2CTransferEventHandler(DRV_I2C_TRANSFER_EVENT event, DRV_I2C_TRANSFER_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* pMyAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_I2C_TRANSFER_EVENT_COMPLETE:
            // This means the data was transferred.
    }
}
```

```

        break;

    case DRV_I2C_TRANSFER_EVENT_ERROR:
        // Error handling here.
        break;

    default:
        break;
}
}

```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine DRV_I2C_Open function.
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_TransferEventHandlerSet
(
    const DRV_HANDLE handle,
    const DRV_I2C_TRANSFER_EVENT_HANDLER eventHandler,
    const uintptr_t context
)

```

c) Data Transfer Functions

DRV_I2C_ReadTransfer Function

This is a blocking function that performs a I2C read operation.

File

[drv_i2c.h](#)

C

```

bool DRV_I2C_ReadTransfer(const DRV_HANDLE handle, uint16_t address, void* const buffer, const
size_t size);

```

Returns

true - read is successful false - error has occurred

Description

This function does a blocking read operation. The function blocks till the read is complete or error has occurred during read. Function will return false to report failure. The failure will occur for the following reasons:

- Invalid input parameters
- Hardware error

Remarks

This function is thread safe in a RTOS application. This function should not be called from an interrupt context. This function is available only in the synchronous mode.

Preconditions

[DRV_I2C_Open](#) must have been called to obtain a valid opened device handle.

Example

```
uint8_t myRxBuffer[MY_RX_BUFFER_SIZE];

// myI2CHandle is the handle returned
// by the DRV\_I2C\_Open function.

// slaveAddress is address of I2C slave device
// to which data is to be written

if (DRV_I2C_ReadTransfer(myI2CHandle, slaveAddress, myRxBuffer, MY_RX_BUFFER_SIZE) == false)
{
    // Error handling here
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine DRV_I2C_Open function.
address	Slave Address
buffer	Destination buffer where read data is stored.
size	Size in bytes of data to be read.

Function

```
bool DRV_I2C_ReadTransfer(
    const DRV\_HANDLE handle,
    uint16_t address,
    void* const buffer,
    const size_t size
)
```

DRV_I2C_ReadTransferAdd Function

Queues a read operation.

File

[drv_i2c.h](#)

C

```
void DRV_I2C_ReadTransferAdd(const DRV\_HANDLE handle, const uint16_t address, void* const
    buffer, const size_t size, DRV\_I2C\_TRANSFER\_HANDLE* const transferHandle);
```

Returns

None

Description

This function schedules a non-blocking read operation. The function returns with a valid transfer handle in the transferHandle argument if the read request was scheduled successfully. The function adds the request to the driver instance transfer 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_I2C_TRANSFER_HANDLE_INVALID](#) in the transferHandle 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 driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a `DRV_I2C_TRANSFER_EVENT_COMPLETE` event if the buffer was processed successfully or `DRV_I2C_TRANSFER_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 I2C Driver Transfer Event Handler that is registered by the client. It should not be called in the event handler associated with another I2C driver instance. It should not be called directly in an ISR. This function is available only in the asynchronous mode.

Preconditions

`DRV_I2C_Open` must have been called to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];
DRV_I2C_TRANSFER_HANDLE transferHandle;

// myI2CHandle is the handle returned
// by the DRV_I2C_Open function.

// slaveAddress is address of I2C slave device
// to which data is to be written

DRV_I2C_ReadTransferAdd(myI2CHandle, slaveAddress, myBuffer, MY_BUFFER_SIZE, &transferHandle);

if(transferHandle == DRV_I2C_TRANSFER_HANDLE_INVALID)
{
    // Error handling here
}

// Event is received when the buffer is processed.
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine <code>DRV_I2C_Open</code> function.
address	Slave address
buffer	buffer where the read data will be stored.
size	Transfer size in bytes.
transferHandle	Pointer to an argument that will contain the return transfer handle. This is <code>DRV_I2C_TRANSFER_HANDLE_INVALID</code> if the request was not successful.

Function

```
void DRV_I2C_ReadTransferAdd(
    const DRV_HANDLE handle,
    const uint16_t address,
    void * const buffer,
    const size_t size,
    DRV_I2C_TRANSFER_HANDLE * const transferHandle
)
```

DRV_I2C_WriteTransfer Function

This is a blocking function that performs a I2C write operation.

File

[drv_i2c.h](#)

C

```
bool DRV_I2C_WriteTransfer(const DRV_HANDLE handle, uint16_t address, void* const buffer, const size_t size);
```

Returns

true - write is successful false - error has occurred

Description

This function does a blocking write operation. The function blocks till the write is complete or error has occurred during write. Function will return false to report failure. The failure will occur for the following reasons:

- Invalid input parameters

Remarks

This function is thread safe in a RTOS application. This function should not be called from an interrupt context. This function is available only in the synchronous mode.

Preconditions

[DRV_I2C_Open](#) must have been called to obtain a valid opened device handle.

Example

```
uint8_t myTxBuffer[MY_TX_BUFFER_SIZE];

// myI2CHandle is the handle returned
// by the DRV_I2C_Open function.

// slaveAddress is address of I2C slave device
// to which data is to be written

if (DRV_I2C_WriteTransfer(myI2CHandle, slaveAddress, myTxBuffer, MY_TX_BUFFER_SIZE) == false)
{
    // Error handling here
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine DRV_I2C_Open function.
address	Slave Address
buffer	Source buffer containing data to be written.
size	Size in bytes of data to be written.

Function

```
bool DRV_I2C_WriteTransfer(
    const DRV_HANDLE handle,
    uint16_t address,
    void* const buffer,
    const size_t size
)
```

DRV_I2C_WriteTransferAdd Function

Queues a write operation.

File

[drv_i2c.h](#)

C

```
void DRV_I2C_WriteTransferAdd(const DRV_HANDLE handle, const uint16_t address, void * const buffer, const size_t size, DRV_I2C_TRANSFER_HANDLE * const transferHandle);
```

Returns

None.

Description

This function schedules a non-blocking write operation. The function returns with a valid transfer handle in the transferHandle argument if the write request was scheduled successfully. The function adds the request to the driver instance transfer 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 transferHandle parameter may be [DRV_I2C_TRANSFER_HANDLE_INVALID](#) for the following reasons:

- if a transfer buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0

If the requesting client registered an event callback with the driver, the driver will issue a [DRV_I2C_TRANSFER_EVENT_COMPLETE](#) event if the buffer was processed successfully or a [DRV_I2C_TRANSFER_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 I2C Driver Transfer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2C driver instance. It should not otherwise be called directly in an ISR. This function is available only in the asynchronous mode.

Preconditions

[DRV_I2C_Open](#) must have been called to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];
DRV_I2C_TRANSFER_HANDLE transferHandle;

// myI2CHandle is the handle returned
// by the DRV_I2C_Open function.

// slaveAddress is address of I2C slave device
// to which data is to be written

DRV_I2C_WriteTransferAdd(myI2CHandle, slaveAddress, myBuffer, MY_BUFFER_SIZE, &transferHandle);

if(transferHandle == DRV_I2C_TRANSFER_HANDLE_INVALID)
{
    // Error handling here
}

// Event is received when the buffer is processed.
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine DRV_I2C_Open function.
address	Slave address
buffer	Data to be written.
size	Transfer size in bytes.
transferHandle	Pointer to an argument that will contain the return transfer handle. This will be DRV_I2C_TRANSFER_HANDLE_INVALID if the function was not successful.

Function

void DRV_I2C_WriteTransferAdd(

```

const     DRV_HANDLE handle,
const uint16_t address,
void * const buffer,
const size_t size,
DRV_I2C_TRANSFER_HANDLE * const transferHandle
)

```

DRV_I2C_WriteReadTransfer Function

This is a blocking function that performs a I2C write followed by a I2C read operation.

File

[drv_i2c.h](#)

C

```
bool DRV_I2C_WriteReadTransfer(const DRV_HANDLE handle, uint16_t address, void* const
writeBuffer, const size_t writeSize, void* const readBuffer, const size_t readSize);
```

Returns

true - transfer is successful false - error has occurred

Description

This function does a blocking write and read operation. The function blocks till the write and read is complete or error has occurred during data transfer. Function will return false to report failure. The failure will occur for the following reasons:

- Invalid input parameters
- Hardware error

Remarks

This function is thread safe in a RTOS application. This function should not be called from an interrupt context. This function is available only in the synchronous mode.

Preconditions

[DRV_I2C_Open](#) must have been called to obtain a valid opened device handle.

Example

```

uint8_t myTxBuffer[MY_TX_BUFFER_SIZE];
uint8_t myRxBuffer[MY_RX_BUFFER_SIZE];

// myI2CHandle is the handle returned
// by the DRV_I2C_Open function.

// slaveAddress is address of I2C slave device
// to which data is to be written

if (DRV_I2C_WriteReadTransfer(myI2CHandle, slaveAddress, myTxBuffer, MY_TX_BUFFER_SIZE,
myRxBuffer, MY_RX_BUFFER_SIZE) == false)
{
    // Error handling here
}

```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine DRV_I2C_Open function.
address	Slave Address
writeBuffer	Source buffer containing data to be written.
writeSize	Size in bytes of data to be written.

readBuffer	Destination buffer where read data is stored.
readSize	Size in bytes of data to be read.

Function

```
bool DRV_I2C_WriteReadTransfer (
    const DRV_HANDLE handle,
    uint16_t address,
    void* const writeBuffer,
    const size_t writeSize,
    void* const readBuffer,
    const size_t readSize
)
```

DRV_I2C_WriteReadTransfer Function

Queues a write followed by read operation.

File

[drv_i2c.h](#)

C

```
void DRV_I2C_WriteReadTransferAdd(const DRV_HANDLE handle, const uint16_t address, void * const
    writeBuffer, const size_t writeSize, void * const readBuffer, const size_t readSize,
    DRV_I2C_TRANSFER_HANDLE * const transferHandle);
```

Returns

None.

Description

This function schedules a non-blocking write followed by read operation. The function returns with a valid transfer handle in the transferHandle argument if the write request was scheduled successfully. The function adds the request to the driver instance transfer 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 transferHandle parameter may be [DRV_I2C_TRANSFER_HANDLE_INVALID](#) for the following reasons:

- if a buffer could not be allocated to the request
- if the input write or read buffer pointer is NULL
- if the write or read buffer size is 0

If the requesting client registered an event callback with the driver, the driver will issue a [DRV_I2C_TRANSFER_EVENT_COMPLETE](#) event if the buffer was processed successfully or a [DRV_I2C_TRANSFER_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 I2C Driver Transfer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2C driver instance. It should not otherwise be called directly in an ISR. This function is available only in the asynchronous mode.

Preconditions

[DRV_I2C_Open](#) must have been called to obtain a valid opened device handle.

Example

```
uint8_t myTxBuffer[MY_TX_BUFFER_SIZE];
uint8_t myRxBuffer[MY_RX_BUFFER_SIZE];
DRV_I2C_TRANSFER_HANDLE transferHandle;

// myI2CHandle is the handle returned
// by the DRV_I2C_Open function.
```

```

// slaveAddress is address of I2C slave device
// to which data is to be written

DRV_I2C_WriteReadTransferAdd(myI2CHandle, slaveAddress, myTxBuffer, MY_TX_BUFFER_SIZE,
myRxBuffer, MY_RX_BUFFER_SIZE, &transferHandle);

if(transferHandle == DRV_I2C_TRANSFER_HANDLE_INVALID)
{
    // Error handling here
}

// Event is received when the buffer is processed.

```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine DRV_I2C_Open function.
address	Slave address
writeBuffer	Data to be written.
writeSize	Size of write buffer in bytes.
readBuffer	Buffer where data to be read is stored.
readSize	Size of the read buffer in bytes.
transferHandle	Pointer to an argument that will contain the return transfer handle. This will be DRV_I2C_TRANSFER_HANDLE_INVALID if the function was not successful.

Function

```

void DRV_I2C_WriteReadTransferAdd (
    const DRV_HANDLE handle,
    const uint16_t address,
    void * const writeBuffer,
    const size_t writeSize,
    void * const readBuffer,
    const size_t readSize,
    DRV_I2C_TRANSFER_HANDLE * const transferHandle
)

```

DRV_I2C_TransferStatusGet Function

Returns the status of the write/read/write-read transfer request.

File

[drv_i2c.h](#)

C

```
DRV_I2C_TRANSFER_EVENT DRV_I2C_TransferStatusGet(const DRV_I2C_TRANSFER_HANDLE transferHandle);
```

Returns

The success or error event of the transfer.

Description

This function can be used to poll the status of the queued transfer request if the application doesn't prefer to use the event handler (callback) function to get notified.

Remarks

This function is available only in the asynchronous mode.

Preconditions

`DRV_I2C_Open` must have been called to obtain a valid opened device handle.

Either the `DRV_I2C_ReadTransferAdd`, `DRV_I2C_WriteTransferAdd` or `DRV_I2C_WriteReadTransferAdd` function must have been called and a valid buffer handle returned.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];
DRV_I2C_TRANSFER_HANDLE transferHandle;
DRV_I2C_TRANSFER_EVENT event;

// myI2CHandle is the handle returned
// by the DRV_I2C_Open function.

DRV_I2C_ReadTransferAdd(myI2CHandle, slaveAddress, myBuffer, MY_BUFFER_SIZE, &transferHandle);

if(transferHandle == DRV_I2C_TRANSFER_HANDLE_INVALID)
{
    // Error handling here
}

//Check the status of the transfer
//This call can be used to wait until the transfer is processed.

event = DRV_I2C_TransferStatusGet(transferHandle);
```

Parameters

Parameters	Description
transferHandle	Handle for the buffer of which the processed number of bytes to be obtained.

Function

```
DRV_I2C_TRANSFER_EVENT DRV_I2C_TransferStatusGet(
const DRV_I2C_TRANSFER_HANDLE transferHandle
)
```

DRV_I2C_ErrorGet Function

Gets the I2C hardware errors associated with the client.

File

`drv_i2c.h`

C

```
DRV_I2C_ERROR DRV_I2C_ErrorGet(const DRV_HANDLE handle);
```

Returns

Errors occurred as listed by `DRV_I2C_ERROR`. This function reports I2C errors if occurred.

Description

This function returns the errors associated with the given client. The call to this function also clears all the associated error flags.

Remarks

The driver clears all the errors internally.

Preconditions

`DRV_I2C_Open` must have been called to obtain a valid opened device handle.

Example

```
// 'handle', returned from the DRV_I2C_Open
```

```

if (DRV_I2C_ErrorGet(handle) == DRV_I2C_ERROR_NACK)
{
    //Errors are cleared by the driver, take respective action
    //for the error case.
}

```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine DRV_I2C_Open function.

Function

DRV_I2C_ERROR DRV_I2C_ErrorGet(const [DRV_HANDLE](#) handle)

d) Data Types and Constants

DRV_I2C_TRANSFER_HANDLE Type

Handle identifying a read, write or write followed by read transfer passed to the driver.

File

[drv_i2c.h](#)

C

```
typedef uintptr_t DRV_I2C_TRANSFER_HANDLE;
```

Description

I2C Driver Transfer Handle

A transfer handle value is returned by a call to the [DRV_I2C_ReadTransferAdd](#)/ [DRV_I2C_WriteTransferAdd](#) or [DRV_I2C_WriteReadTransferAdd](#) functions. This handle is associated with the transfer passed into the function and it allows the application to track the completion of the data from (or into) that transfer. The transfer handle value returned from the "transfer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The transfer 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 transfer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_I2C_TRANSFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

[drv_i2c.h](#)

C

```
typedef enum {
    DRV_I2C_TRANSFER_EVENT_PENDING,
    DRV_I2C_TRANSFER_EVENT_COMPLETE,
    DRV_I2C_TRANSFER_EVENT_ERROR
} DRV_I2C_TRANSFER_EVENT;
```

Members

Members	Description
DRV_I2C_TRANSFER_EVENT_PENDING	Transfer request is pending
DRV_I2C_TRANSFER_EVENT_COMPLETE	All data from or to the buffer was transferred successfully.
DRV_I2C_TRANSFER_EVENT_ERROR	There was an error while processing the buffer transfer request.

Description

I2C Driver Transfer Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the [DRV_I2C_ReadTransferAdd](#), [DRV_I2C_WriteTransferAdd](#) or [DRV_I2C_WriteReadTransferAdd](#) 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_I2C_TransferEventHandlerSet](#) function when a buffer transfer request is completed.

DRV_I2C_TRANSFER_EVENT_HANDLER Type

Pointer to a I2C Driver Transfer Event handler function

File

[drv_i2c.h](#)

C

```
typedef void (* DRV_I2C_TRANSFER_EVENT_HANDLER)(DRV_I2C_TRANSFER_EVENT event,  
DRV_I2C_TRANSFER_HANDLE transferHandle, uintptr_t context);
```

Returns

None.

Description

I2C Driver Transfer Event Handler Function Pointer

This data type defines the required function signature for the I2C driver buffer event handling callback function. A client must register a pointer using the 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_I2C_TRANSFER_EVENT_COMPLETE`, it means that the data was transferred successfully.

If the event is `DRV_I2C_TRANSFER_EVENT_ERROR`, it means that the data was not transferred successfully.

The `transferHandle` parameter contains the transfer handle of the transfer that associated with the event. And `transferHandle` will be valid while the transfer request is in the queue and during callback, unless an error occurred. After callback returns, the driver will retire the transfer handle.

The `context` parameter contains the a handle to the client context, provided at the time the event handling function was registered using the [DRV_I2C_TransferEventHandlerSet](#) 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 peripheral's interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The [DRV_I2C_ReadTransferAdd](#), [DRV_I2C_WriteTransferAdd](#) and [DRV_I2C_WriteReadTransferAdd](#) 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, I2C2 driver buffers cannot be added in I2C1 driver event handler.

Example

```
void APP_MyTransferEventHandler( DRV_I2C_TRANSFER_EVENT event,  
                                DRV_I2C_TRANSFER_HANDLE transferHandle,
```

```

        uintptr_t context )
{
    MY_APP_DATA_STRUCT* pAppData = (MY_APP_DATA_STRUCT*) context;

    switch(event)
    {
        case DRV_I2C_TRANSFER_EVENT_COMPLETE:
            // Handle the completed buffer.
            break;

        case DRV_I2C_TRANSFER_EVENT_ERROR:
        default:
            // Handle error.
            break;
    }
}

```

Parameters

Parameters	Description
event	Identifies the type of event
transferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_I2C_TRANSFER_HANDLE_INVALID Macro

Definition of an invalid transfer handle.

File

[drv_i2c.h](#)

C

```
#define DRV_I2C_TRANSFER_HANDLE_INVALID
```

Description

I2C Driver Invalid Transfer Handle

This is the definition of an invalid transfer handle. An invalid transfer handle is returned by [DRV_I2C_ReadTransferAdd](#), [DRV_I2C_WriteTransferAdd](#) and [DRV_I2C_WriteReadTransferAdd](#) functions if the buffer add request was not successful.

Remarks

None

Files

Files

Name	Description
drv_i2c.h	I2C Driver Interface Header File

Description

This section will list only the library's interface header file(s).

drv_i2c.h

I2C Driver Interface Header File

Enumerations

	Name	Description
	DRV_I2C_TRANSFER_EVENT	Identifies the possible events that can result from a buffer add request.

Functions

	Name	Description
≡	DRV_I2C_Close	Closes an opened-instance of the I2C driver.
≡	DRV_I2C_ErrorGet	Gets the I2C hardware errors associated with the client.
≡	DRV_I2C_Initialize	Initializes the I2C instance for the specified driver index.
≡	DRV_I2C_Open	Opens the specified I2C driver instance and returns a handle to it.
≡	DRV_I2C_ReadTransfer	This is a blocking function that performs a I2C read operation.
≡	DRV_I2C_ReadTransferAdd	Queues a read operation.
≡	DRV_I2C_Status	Gets the current status of the I2C driver module.
≡	DRV_I2C_TransferEventHandlerSet	Allows a client to identify a transfer event handling function for the driver to call back when queued transfers have finished.
≡	DRV_I2C_TransferStatusGet	Returns the status of the write/read/write-read transfer request.
≡	DRV_I2C_WriteReadTransfer	This is a blocking function that performs a I2C write followed by a I2C read operation.
≡	DRV_I2C_WriteReadTransferAdd	Queues a write followed by read operation.
≡	DRV_I2C_WriteTransfer	This is a blocking function that performs a I2C write operation.
≡	DRV_I2C_WriteTransferAdd	Queues a write operation.

Macros

	Name	Description
	DRV_I2C_TRANSFER_HANDLE_INVALID	Definition of an invalid transfer handle.

Types

	Name	Description
	DRV_I2C_TRANSFER_EVENT_HANDLER	Pointer to a I2C Driver Transfer Event handler function
	DRV_I2C_TRANSFER_HANDLE	Handle identifying a read, write or write followed by read transfer passed to the driver.

Description

I2C Driver Interface Header File

The I2C device driver provides a simple interface to manage the I2C modules on Microchip PIC32 microcontrollers. This file provides the interface definition for the I2C driver.

File Name

drv_i2c.h

Company

Microchip Technology Inc.

I2S Driver Library Help

This section describes the I2S Driver Library.

Introduction

This library provides an interface to manage the I2S Audio Protocol Interface Modes.

Description

The I2S Driver is connected to a hardware module that provides the actual I2S stream, on some MCUs this is a Serial Peripheral Interface (SPI), on others it may be an I2S Controller (I2SC), or Serial Synchronous Controller (SSC).

The I2S hardware peripheral is then interfaced to various devices such as codecs and Bluetooth modules to provide microcontroller-based audio solutions.

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 [What is MPLAB Harmony?](#) section for how the driver interacts with the framework.

Example Applications:

This library is used by the following applications, among others:

- [audio/apps/audio_tone](#)
- [audio/apps/audio_tone_linkeddma](#)
- [audio/apps/microphone_loopback](#)

Abstraction Model

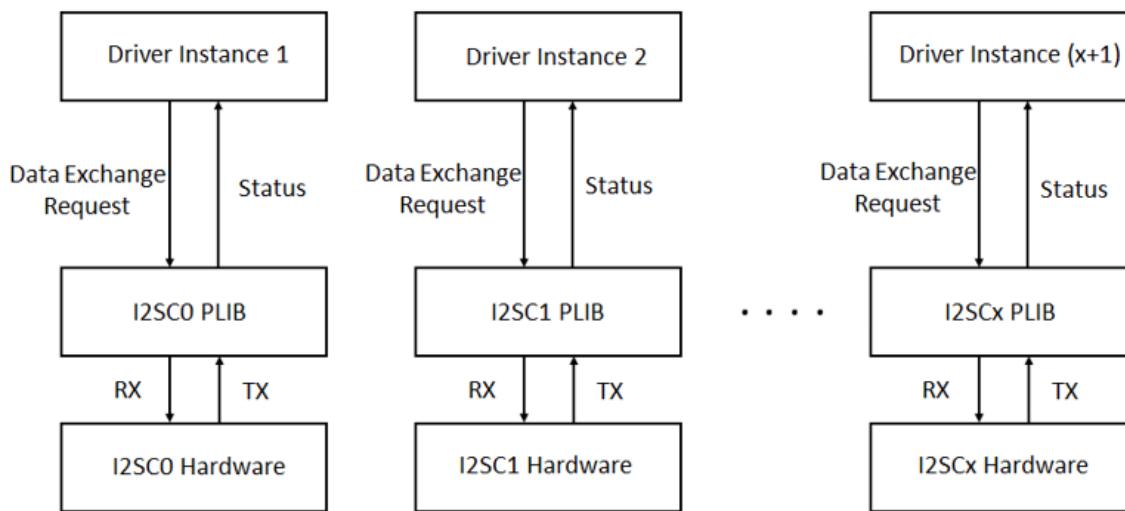
The I2S Driver provides a high level abstraction of the lower level (SPI/I2SC/SSC) I2S modules with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the I2S Driver Library interface.

Description

Different types of I2S capable PLIBs are available on various Microchip microcontrollers. Some have an internal buffer mechanism and some do not. The buffer depth varies across part families. The I2S Driver Library abstracts out these differences and provides a unified model for audio data transfer across different types of I2S 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 abstraction model used by the I2S Driver Library. The I2SC Peripheral is used as an example of an I2S-capable PLIB.

I2S Driver Abstraction Model



The PLIBs currently provided, such as SSC and I2SC, only support an interrupt/DMA mode of operation. Polled mode of operation is not supported.

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 device initialization 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 get error functions, L/R clock sync, 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.

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;

/* I2S Driver Initialization Data */
DRV_I2S_INIT drvI2S0InitData =
{
    .i2sPlib = &drvI2S0PlibAPI,
    .interruptI2S = DRV_I2S_INT_SRC_IDX0,
    .numClients = DRV_I2S_CLIENTS_NUMBER_IDX0,
    .queueSize = DRV_I2S_QUEUE_SIZE_IDX0,
    .dmaChannelTransmit = DRV_I2S_XMIT_DMA_CH_IDX0,
    .dmaChannelReceive = DRV_I2S_RCV_DMA_CH_IDX0,
    .i2sTransmitAddress = (void *)&(SSC_REGS->SSC_THR),
    .i2sReceiveAddress = (void *)&(SSC_REGS->SSC_RHR),
    .interruptDMA = XDMAC IRQn,
    .dmaDataLength = DRV_I2S_DATA_LENGTH_IDX0,
};

sysObj.drvI2S0 = DRV_I2S_Initialize(DRV_I2S_INDEX_0, (SYS_MODULE_INIT *)&drvI2S0InitData);
```

Task Routine

There is no task routine, since polled mode is not currently supported.

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.

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(drvObj->i2sDriverModuleIndex,
```

```

(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

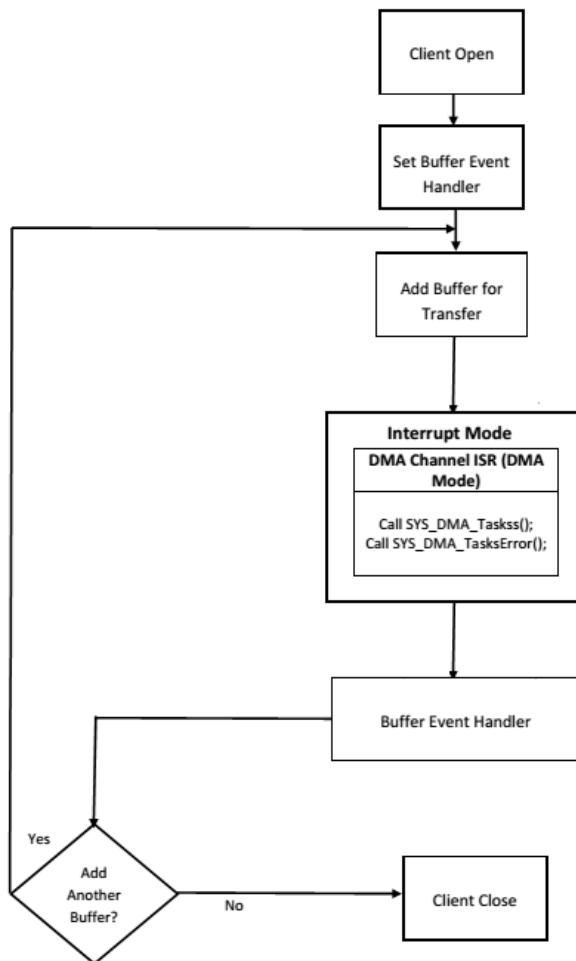
This section provides information on buffered client operations.

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 from the I2S channel ISR in interrupt mode.

The following diagram illustrates the buffered data operations





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

Note:

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. When the DMA Channel has finished, the callback function registered in step 8 will be called.
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:

```

// 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_WriteBufferAdd(handle, myAudioBuffer, BUFFER_SIZE,
        &bufferHandle);
    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 ( );
}
```

Client Operations - Non-buffered

This section provides information on non-buffered client operations.

Description

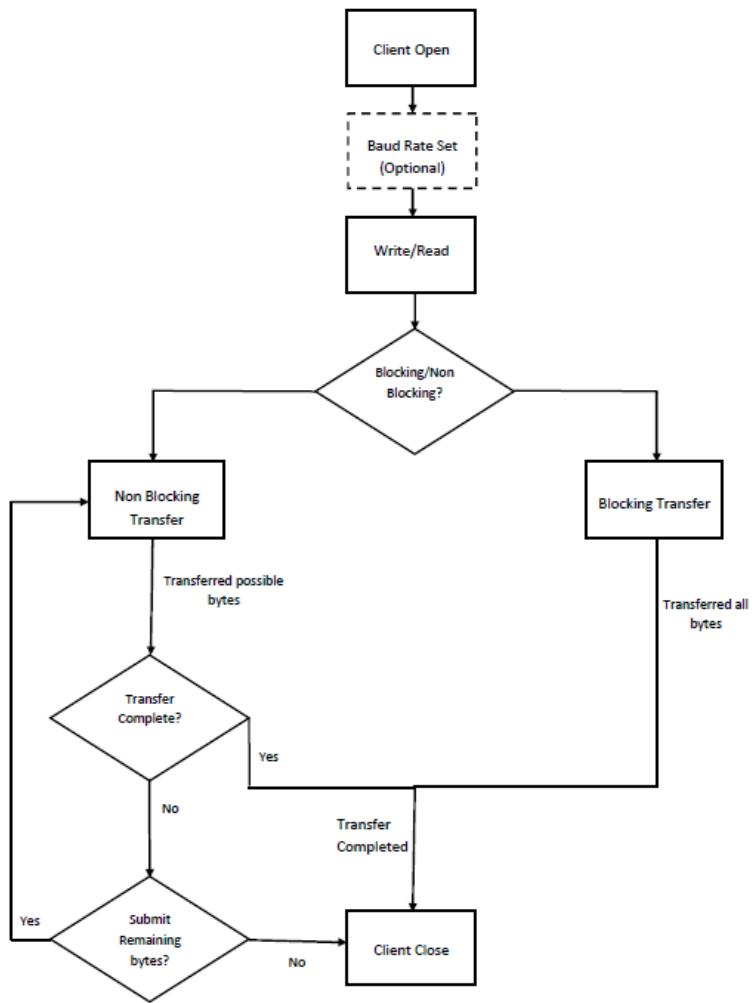
Client Operations - Non-buffered

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

The configuration of the I2S Driver Library is based on the file `configurations.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 Help section for more details.

System Configuration

Configurations for driver instances, polled/interrupt mode, etc.

Configuring MHC

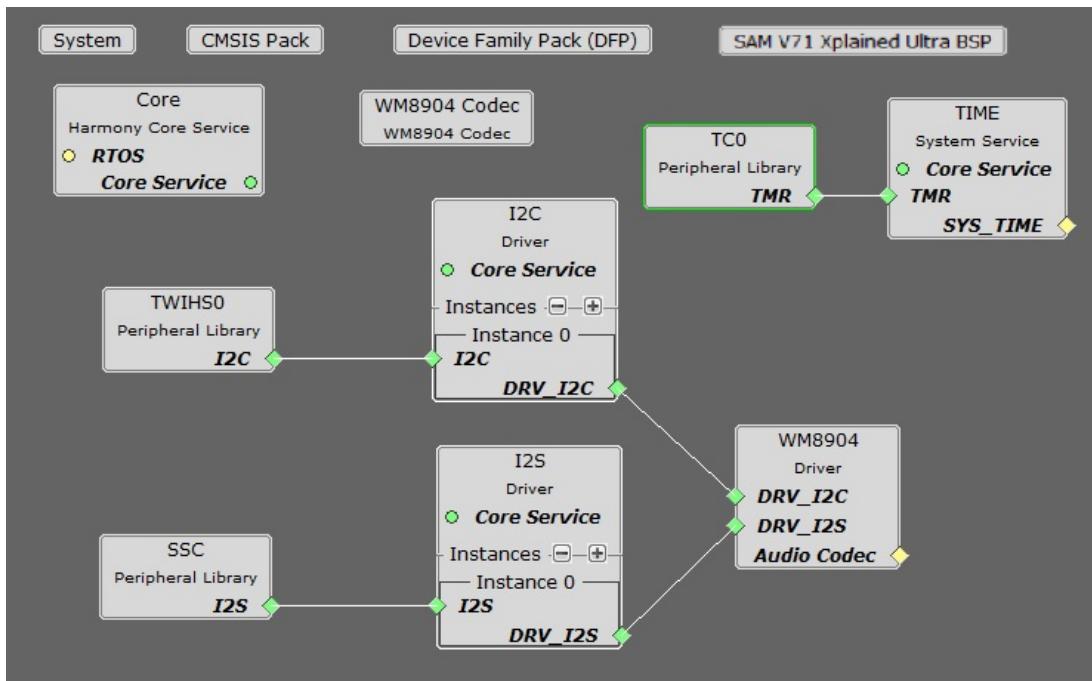
Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

Description

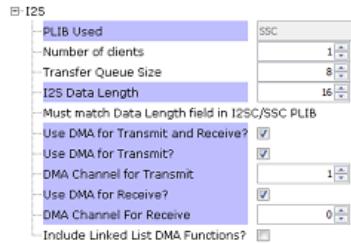
When building a new application, start by creating a 32-bit MPLAB Harmony 3 project in MPLAB X IDE by selecting *File > New Project*. Choose the Configuration name based on the BSP, and select the appropriate processor (such as ATSAME70Q21B).

In MHC, under Available Components select the appropriate BSP, such as SAM E70 Xplained Ultra. Under Audio->Templates, double-click on a codec template such as WM8904. Answer Yes to all questions.

You should end up with a project graph that looks like this, after rearranging the boxes:



Click on the I2S Driver component, Instance 0, and the following menu will be displayed in the Configurations Options:



PLIB Used will display the hardware peripheral instance connected to the I2S Driver, such as SPI0, SSC, or I2SC1.

Number of Clients indicates the maximum number of clients that can be connected to the I2S Driver.

Transfer Queue Size indicates number of buffers, of each transfer queue (transmit/receive).

I2S Data Length is the number of bits for one channel of audio (left or right). It must match the size of the PLIB.

Use DMA for Transmit and Receive should always be checked if using DMA, which is currently the only supported mode.

Use DMA for Transmit should be checked if sending data to a codec or Bluetooth module.

DMA Channel for Transmit indicates the DMA channel # assigned (done automatically when you connect the PLIB).

Use DMA for Receive should be checked if receiving data from a codec or Bluetooth module. However if you are only writing to the I2S stream, leaving this checked won't harm anything.

DMA Channel for Receive indicates the DMA channel # assigned (done automatically when you connect the PLIB).

Included Linked List DMA Functions should be checked if using the Linked DMA feature of some MCUs.

You can also bring in the I2S Driver by itself, by double clicking I2S under Harmony->Drivers in the Available Components list.

You will then need to add any additional needed components manually and connect them together.

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 core/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 (generated via template core/driver/i2s/templates/drv_i2s.h.ft1)

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

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_i2s.c	This file contains the core implementation of the I2S driver with DMA support (generated via template core/driver/i2s/templates/drv_i2s.c.ft1)

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	

Module Dependencies

The I2S Driver Library depends on the following modules:

- SPI Peripheral Library, or
- SSC Peripheral Library
- I2SC Peripheral Library

Library Interface

e) Data Types and Constants

	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_I2S_INIT	Defines the data required to initialize the I2S driver
	DRV_BAUDSET	This is type DRV_BAUDSET.
	DRV_I2S_DMA_WIDTH	This is type DRV_I2S_DMA_WIDTH.
	DRV_I2S_PLIB_INTERFACE	Defines the data required to initialize the I2S driver PLIB Interface.
	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.
	DRV_I2S_ERROR	Defines the data required to setup the I2S transfer
	DRV_HANDLE_INVALID	Invalid device handle.
	MAIN_RETURN_CODES	Defines return codes for "main".
	SYS_MODULE_INIT	Initializes a module (including device drivers) as requested by the system.
	DRV_I2S_LRCLK_GET	This is type DRV_I2S_LRCLK_GET.

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**b) Client Setup Functions****c) Data Transfer Functions****d) Miscellaneous Functions****e) Data Types and Constants****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_I2S_INIT Structure

Defines the data required to initialize the I2S driver

File

[drv_i2s_definitions.h](#)

C

```
typedef struct {
    DRV_I2S_PLIB_INTERFACE * i2sPlib;
    INT_SOURCE interruptI2S;
    uintptr_t clientObjPool;
    size_t numClients;
    uintptr_t transferObjPool;
    size_t queueSize;
    SYS_DMA_CHANNEL dmaChannelTransmit;
    SYS_DMA_CHANNEL dmaChannelReceive;
    void * i2sTransmitAddress;
    void * i2sReceiveAddress;
    INT_SOURCE interruptDMA;
    uint8_t dmaDataLength;
} DRV_I2S_INIT;
```

Members

Members	Description
DRV_I2S_PLIB_INTERFACE * i2sPlib;	Identifies the PLIB API set to be used by the driver to access the <ul style="list-style-type: none"> peripheral.
INT_SOURCE interruptI2S;	Interrupt source ID for the I2S interrupt.
uintptr_t clientObjPool;	Memory Pool for Client Objects

size_t numClients;	Number of clients
uintptr_t transferObjPool;	Queue for Transfer Objects
size_t queueSize;	Driver Queue Size
SYS_DMA_CHANNEL dmaChannelTransmit;	I2S transmit DMA channel.
SYS_DMA_CHANNEL dmaChannelReceive;	I2S receive DMA channel.
void * i2sTransmitAddress;	I2S transmit register address used for DMA operation.
void * i2sReceiveAddress;	I2S receive register address used for DMA operation.
INT_SOURCE interruptDMA;	Interrupt source ID for DMA interrupt.
uint8_t dmaDataLength;	DMA data length from I2S or SSC PLIB

Description

I2S Driver Initialization Data

This data type defines the data required to initialize or the I2S driver.

Remarks

None.

DRV_BAUDSET Type

File

[drv_i2s_definitions.h](#)

C

```
typedef void (* DRV_BAUDSET)(const uint32_t);
```

Description

This is type DRV_BAUDSET.

DRV_I2S_DMA_WIDTH Enumeration

File

[drv_i2s_definitions.h](#)

C

```
typedef enum {
    DRV_I2S_DMA_WIDTH_8_BIT = 0,
    DRV_I2S_DMA_WIDTH_16_BIT = 1
} DRV_I2S_DMA_WIDTH;
```

Description

This is type DRV_I2S_DMA_WIDTH.

DRV_I2S_PLIB_INTERFACE Structure

Defines the data required to initialize the I2S driver PLIB Interface.

File

[drv_i2s_definitions.h](#)

C

```
typedef struct {
    DRV_BAUDSET setBaud;
    DRV_I2S_LRCLK_GET I2S_LRCLK_Get;
} DRV_I2S_PLIB_INTERFACE;
```

Members

Members	Description
DRV_BAUDSET setBaud;	I2S PLIB baud rate set function

Description

I2S Driver PLIB Interface Data

This data type defines the data required to initialize the I2S driver PLIB Interface.

Remarks

None.

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.

DRV_I2S_ERROR Enumeration

Defines the data required to setup the I2S transfer

File

[drv_i2s_definitions.h](#)

C

```

typedef enum {
    DRV_I2S_ERROR_NONE = 0,
    DRV_I2S_ERROR_OVERRUN = 1
} DRV_I2S_ERROR;

```

Description

I2S Driver Setup Data

This data type defines the data required to setup the I2S transfer. The data is passed to the DRV_I2S_TransferSetup API to setup

the I2S peripheral settings dynamically.

Remarks

None.

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.

MAIN_RETURN_CODES Enumeration

Defines return codes for "main".

File

[system_common.h](#)

C

```
typedef enum {
    MAIN_RETURN_FAILURE = -1,
    MAIN_RETURN_SUCCESS = 0
} MAIN_RETURN_CODES;
```

Description

Main Routine Codes Enumeration

This enumeration provides a predefined list of return codes for the main function. These codes can be passed into the [MAIN_RETURN_CODE](#) macro to convert them to the appropriate type (or discard them if not needed) for the Microchip C-language compiler in use.

Remarks

The main function return type may change, depending upon which family of Microchip microcontrollers is chosen. Refer to the user documentation for the C-language compiler in use for more information.

Example

```
MAIN_RETURN main ( void )
{
    SYS_Initialize(...);

    while(true)
    {
        SYS_Tasks();
    }
}
```

```

    return MAIN_RETURN_CODE(MAIN_RETURN_SUCCESS);
}

```

SYS_MODULE_INIT Union

Initializes a module (including device drivers) as requested by the system.

File

[system_module.h](#)

C

```

typedef union {
    uint8_t value;
    struct {
        uint8_t reserved : 4;
    } sys;
} SYS_MODULE_INIT;

```

Members

Members	Description
<code>uint8_t reserved : 4;</code>	Module-definable field, module-specific usage

Description

System Module Init

This structure provides the necessary data to initialize or reinitialize a module (including device drivers). The structure can be extended in a module specific way as to carry module specific initialization data.

Remarks

This structure is used in the device driver routines DRV__Initialize and DRV__Reinitialize that are defined by each device driver.

DRV_I2S_LRCLK_GET Type

File

[drv_i2s_definitions.h](#)

C

```

typedef uint32_t (* DRV_I2S_LRCLK_GET)();

```

Description

This is type DRV_I2S_LRCLK_GET.

Files

Files

Name	Description
driver_common.h	This file defines the common macros and definitions used by the driver definition and implementation headers.
drv_i2s.h	This is file drv_i2s.h.
drv_i2s_definitions.h	I2S Driver Definitions Header File
system.h	Driver layer data types and definitions.
sys_dma.h	This is file sys_dma.h.

Description

drv_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_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

None.

File Name

drv_common.h

Company

Microchip Technology Inc.

drv_i2s.h

This is file *drv_i2s.h*.

drv_i2s_definitions.h

I2S Driver Definitions Header File

Enumerations

	Name	Description
	DRV_I2S_DMA_WIDTH	This is type DRV_I2S_DMA_WIDTH .
	DRV_I2S_ERROR	Defines the data required to setup the I2S transfer

Structures

	Name	Description
	DRV_I2S_INIT	Defines the data required to initialize the I2S driver
	DRV_I2S_PLIB_INTERFACE	Defines the data required to initialize the I2S driver PLIB Interface.

Types

	Name	Description
	DRV_BAUDSET	This is type DRV_BAUDSET.
	DRV_I2S_LRCLK_GET	This is type DRV_I2S_LRCLK_GET.

Description

SPI Driver Definitions Header File

This file provides implementation-specific definitions for the I2S driver's system interface.

File Name

`drv_i2s_definitions.h`

Company

Microchip Technology Inc.

system.h

Driver layer data types and definitions.

Description

Driver Layer Interface Header

This file defines the common macros and definitions for the driver layer modules.

Remarks

The parent directory to the "system" directory should be added to the compiler's search path for header files such that the following include statement will successfully include this file.

`#include "system/system.h"`

File Name

`driver.h`

Company

Microchip Technology Inc.

sys_dma.h

This is file `sys_dma.h`.

Memory Driver Library Help

This section describes the Memory Driver Library.

Introduction

This Library provides an interface to use Memory Driver to communicate with various media devices like QSPI_FLASH, EEPROM_FLASH, NVM_FLASH etc..

Description

Memory Driver is a multi-client multi-instance buffer model based block driver interface which can be used to communicate with various media devices.

Key Features of Memory Driver:

1. It works in both Asynchronous and Synchronous modes of operation.
2. It works in both Bare Metal and RTOS environment.
3. It can be used to communicate with Different Media's via common Media Interface.
 1. Media which can be accessed directly by Peripheral Libraries (NVM_PLIB).
 2. Media which can be accessed by protocol drivers which in-turn connect to respective peripheral Libraries (QSPI_FLASH_DRIVER->QSPI_PLIB)

Using the Library

This topic describes the basic architecture of the Memory Driver and provides information on how it works.

Description

Memory Driver can be used to communicate with various media devices using common interface functions. The library interface functions can be used in 2 different modes.

Modes supported:

1. Asynchronous mode - Bare-metal and RTOS
2. Synchronous - RTOS

It can be used in multiple ways

1. Application can directly use the Memory driver API's to perform Media operations.
2. Application can use File System service layer and perform file operations.
3. It can be interfaced with Middleware's like USB.

Abstraction Model

Provides abstraction of the Memory Driver Library with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

Memory Driver provides abstraction to communicate with different media devices via a common Media Interface (MEMORY_DEVICE_API).

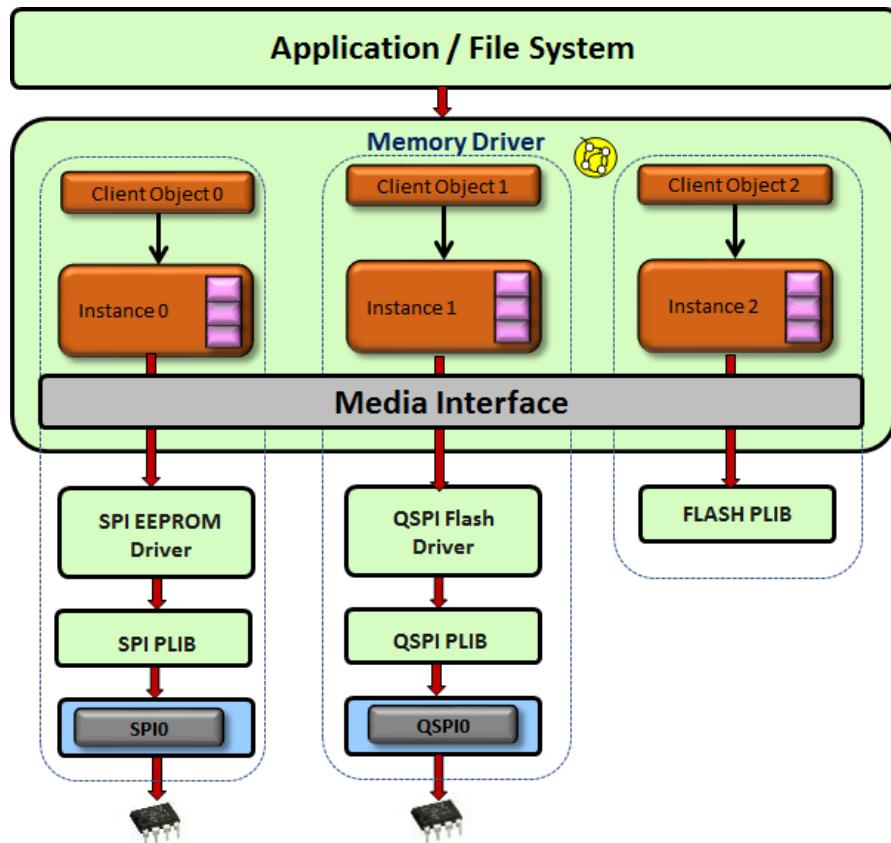


Figure-1 Memory Driver Block Diagram

How the Library Works

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

Description

Memory Library is a multi-client, multi-instance buffer queue model based block driver interface.

Memory Driver Common Features

- Each instance of the driver has its own buffer queue which allows the capability to not block other media operations.
- Each instance of the driver can either have File-system as client or Application as client.
- Every transfer request expects data in blocks. Block details (**Size and number of blocks**) can be retrieved by [DRV_MEMORY_GeometryGet\(\)](#)
- Driver provides feature to register call back for transfer complete event, which can be used by clients to get notified.
- When Memory driver is connected to File System buffer queue is disabled as File system is blocking interface.

Overview on behavior of memory driver in below 2 modes:

Asynchronous Mode	Synchronous Mode
Works in both Bare-Metal and RTOS environment	Works only in RTOS Environment
Provides Non-Blocking behavior.	Provides Blocking behavior. Application thread gets blocked on a semaphore until transfer request is completed.
API's to be used <code>DRV_MEMORY_AsyncXxx()</code>	API's to be used <code>DRV_MEMORY_SyncXxx()</code>
API's return with a valid handle which can be used to check whether transfer request is accepted	API's return true or false to indicate whether the whole transfer is completed.

Bare-Metal: A dedicated task routine <code>DRV_MEMORY_Tasks()</code> is called from <code>SYS_Tasks()</code> to process the data from the instance queue	As the Driver works in complete blocking behavior there is no task routine generated.
RTOS: A dedicated thread is created for task routine <code>DRV_MEMORY_Tasks()</code> to process the data from the instance queue	
A Client specific handler will be called to indicate the status of transfer.	A Client specific handler will be called to indicate the status of transfer before returning from API. Although the return type of API also can be used to identify the status of transfer.

How to plugin Media's to Memory Driver

The `DRV_MEMORY_INIT` data structure allows a Media driver Or Media PLIB to be plugged into the Memory Block Driver. Any media that needs to be plugged into the Memory Block Driver needs to implement the interface (function pointer signatures) specified by the `MEMORY_DEVICE_API` type.

```

static uint8_t gDrvMemory0EraseBuffer[DRV_SST26_ERASE_BUFFER_SIZE] __attribute__((aligned(32))) ;

static DRV_MEMORY_CLIENT_OBJECT gDrvMemory0ClientObject[DRV_MEMORY_CLIENTS_NUMBER_IDX0] = { 0 } ;

static DRV_MEMORY_BUFFER_OBJECT gDrvMemory0BufferObject[DRV_MEMORY_BUFFER_QUEUE_SIZE_IDX0] = { 0 } ;

const MEMORY_DEVICE_API drvMemory0DeviceAPI = {
    .Open          = DRV_SST26_Open,
    .Close         = DRV_SST26_Close,
    .Status        = DRV_SST26_Status,
    .SectorErase   = DRV_SST26_SectorErase,
    .Read          = DRV_SST26_Read,
    .PageWrite     = DRV_SST26_PageWrite,
    .EventHandlerSet = NULL,
    .GeometryGet   = (GEOMETRY_GET)DRV_SST26_GeometryGet,
    .TransferStatusGet = (TRANSFER_STATUS_GET)DRV_SST26_TransferStatusGet
};

const DRV_MEMORY_INIT drvMemory0InitData =
{
    .memDevIndex      = DRV_SST26_INDEX,
    .memoryDevice     = &drvMemory0DeviceAPI,
    .isMemDevInterruptEnabled = false,
    .isFsEnabled       = true,
    .deviceMediaType  = (uint8_t)SYS_FS_MEDIA_TYPE_SPIFLASH,
    .ewBuffer          = &gDrvMemory0EraseBuffer[0],
    .clientObjPool    = (uintptr_t)&gDrvMemory0ClientObject[0],
    .bufferObj        = (uintptr_t)&gDrvMemory0BufferObject[0],
    .queueSize         = DRV_MEMORY_BUFFER_QUEUE_SIZE_IDX0,
    .nClientsMax      = DRV_MEMORY_CLIENTS_NUMBER_IDX0
};

sysObj.drvMemory0 = DRV_MEMORY_Initialize((SYS_MODULE_INDEX)DRV_MEMORY_INDEX_0,
(SYS_MODULE_INIT *)&drvMemory0InitData);

```

Configuring the Library

This Section provides information on how to configure Memory Driver library.

Description

Memory Driver library should be configured via MHC. Below are the Snapshot of the MHC configuration window for Memory driver

and brief description.

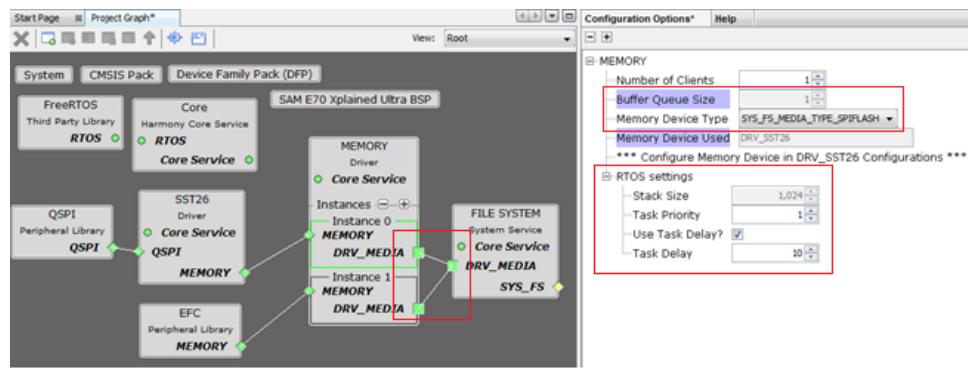


Figure-2 Asynchronous Mode with RTOS and File-System

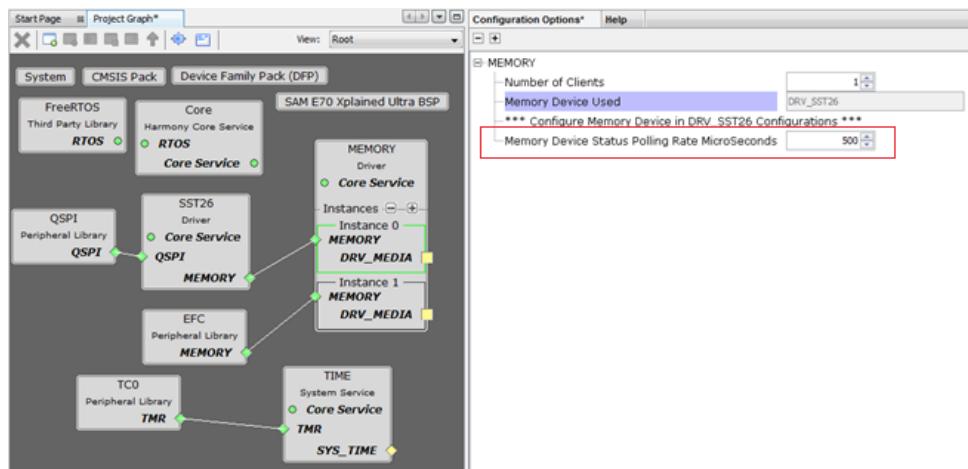


Figure-3 Synchronous Mode with RTOS and without File System

Common User Configuration for all Instances

1. Driver Mode

1. Allows User to select the mode of driver(**Asynchronous or Synchronous**). This setting is common for all the instances

Instance Specific User Configurations

1. Number Of Clients

1. Specifies number of clients to access the specific instance of the driver.

2. Buffer Queue Size

1. Specifies maximum number of requests to be buffered in queue.
2. When Connected to File system this value is set to 1 and non modifiable

3. **This Configuration is displayed only in Asynchronous mode. Refer Figure-2**

3. Memory Device Type

1. Specifies the type of the device connected when File system is connected to this instance. **Refer Figure-2**

4. Memory Device Used

1. Specifies the Media device connected.

5. Memory Device Status Polling Rate Microseconds

1. Specifies the interval to poll for the transfer status.

2. **Option is Displayed only in Synchronous mode and when Memory Device Used does not support interrupt mode. Refer Figure-3**

6. RTOS Settings

1. This Configuration is displayed only in Asynchronous mode. Refer Figure-2
2. Stack Size - Read Only
3. Task Priority
 1. Specifies Priority for the Memory driver task thread
4. Task Delay
 1. Specifies Delay to sleep for the Memory task Thread after every call.

Building the Library

This Section provides information on how the Memory Driver Library can be built.

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Functions

	Name	Description
≡◊	DRV_MEMORY_Initialize	Initializes the Memory instance for the specified driver index
≡◊	DRV_MEMORY_Status	Gets the current status of the Memory driver module.
≡◊	DRV_MEMORY_Tasks	Maintains the Memory driver's internal state machine.

b) Core Client Functions

	Name	Description
≡◊	DRV_MEMORY_Open	Opens the specified Memory driver instance and returns a handle to it
≡◊	DRV_MEMORY_Close	Closes an opened-instance of the Memory driver
≡◊	DRV_MEMORY_TransferHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

c) Block Operation Functions

	Name	Description
≡◊	DRV_MEMORY_AsyncErase	Erase the specified number of memory blocks from the specified block start. Each block is equal to sector size of the memory device attached.
≡◊	DRV_MEMORY_AsyncEraseWrite	Erase and Write blocks of data in the sectors where the block start belongs.
≡◊	DRV_MEMORY_AsyncRead	Reads nblocks of data from the specified block start.
≡◊	DRV_MEMORY_AsyncWrite	Writes nblocks of data starting at the specified block start.
≡◊	DRV_MEMORY_SyncErase	Erase the specified number of memory blocks from the specified block start. Each block is equal to sector size of the memory device attached.
≡◊	DRV_MEMORY_Erase	This routine provides interface to the file system to perform a media erase operation in synchronous mode of the Memory driver.
≡◊	DRV_MEMORY_SyncEraseWrite	Erase and Write blocks of data in the sectors where the block start belongs.
≡◊	DRV_MEMORY_EraseWrite	This routine provides interface to the file system to perform a media erase-write operation in synchronous mode of the Memory driver.
≡◊	DRV_MEMORY_SyncRead	Reads nblock of data from the specified block start.
≡◊	DRV_MEMORY_Read	This routine provides interface to the file system to perform a media read operation in synchronous mode of the Memory driver.
≡◊	DRV_MEMORY_SyncWrite	Writes nblock of data starting at the specified block start.

≡	DRV_MEMORY_Write	This routine provides interface to the file system to perform a media write operation in synchronous mode of the Memory driver.
≡	DRV_MEMORY_CommandStatusGet	Gets the current status of the command.
≡	DRV_MEMORY_TransferStatusGet	Gets the current status of the transfer request on attached device.

d) Media Interface Functions

	Name	Description
≡	DRV_MEMORY_AddressGet	Returns the Memory media start address
≡	DRV_MEMORY_GeometryGet	Returns the geometry of the memory device.
≡	DRV_MEMORY_IsAttached	Returns the physical attach status of the Media.
≡	DRV_MEMORY_IsWriteProtected	Returns the write protect status of the Memory.

e) Data Types and Constants

	Name	Description
	DRV_MEMORY_COMMAND_HANDLE	Handle to identify commands queued in the driver.
	DRV_MEMORY_COMMAND_HANDLE_INVALID	This value defines the Memory Driver's Invalid Command Handle.
	DRV_MEMORY_COMMAND_STATUS	Memory Driver command Status
	DRV_MEMORY_DEVICE_INTERFACE	Memory Device API.
	DRV_MEMORY_DEVICE_OPEN	Function pointer typedef to open the attached media
	DRV_MEMORY_DEVICE_CLOSE	Function pointer typedef to close the attached media
	DRV_MEMORY_DEVICE_SECTOR_ERASE	Function pointer typedef to erase a sector from attached media
	DRV_MEMORY_DEVICE_STATUS	Function pointer typedef to get the status of the attached media
	DRV_MEMORY_DEVICE_READ	Function pointer typedef to read from the attached media
	DRV_MEMORY_DEVICE_PAGE_WRITE	Function pointer typedef to write a page to the attached media
	DRV_MEMORY_DEVICE_EVENT_HANDLER_SET	Function pointer typedef to set the event handler with attached media
	DRV_MEMORY_DEVICE_GEOMETRY_GET	Function pointer typedef to get the Geometry details from attached media
	DRV_MEMORY_DEVICE_TRANSFER_STATUS_GET	Function pointer typedef to get the transfer Status from attached media
	DRV_MEMORY_EVENT	Identifies the possible events that can result from a request.
	DRV_MEMORY_EVENT_HANDLER	Function pointer typedef for event handler to be sent to attached media
	DRV_MEMORY_INIT	Memory Driver Initialization Data
	DRV_MEMORY_TRANSFER_HANDLER	Pointer to a Memory Driver Event handler function
	MEMORY_DEVICE_GEOMETRY	Memory Device Geometry Table.
	MEMORY_DEVICE_TRANSFER_STATUS	Memory Device Transfer Status.

Description

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

Refer to each section for a detailed description.

a) System Functions

DRV_MEMORY_Initialize Function

Initializes the Memory instance for the specified driver index

File

[drv_memory.h](#)

C

```
SYS_MODULE_OBJ DRV_MEMORY_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 Memory 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 Memory routine is called.

This routine should only be called once during system initialization.

This routine will NEVER block for hardware access. If the operation requires time to allow the hardware to initialize, it will be reported by the [DRV_MEMORY_Status](#) operation. The system must use [DRV_MEMORY_Status](#) to find out when the driver is in the ready state.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the Memory Driver  
// with SST26 serial flash device attached and File system Enabled.  
  
SYS_MODULE_OBJ objectHandle;  
  
static uint8_t gDrvMemory0EraseBuffer[DRV_MEMORY_ERASE_BUFFER_SIZE_IDX0]  
__attribute__((aligned(32)));  
  
static DRV_MEMORY_CLIENT_OBJECT gDrvMemory0ClientObject[DRV_MEMORY_CLIENTS_NUMBER_IDX0] = { 0 };  
  
static DRV_MEMORY_BUFFER_OBJECT gDrvMemory0BufferObject[DRV_MEMORY_BUFFER_QUEUE_SIZE_IDX0] = {  
0 };  
  
const DRV_MEMORY_DEVICE_INTERFACE drvMemory0DeviceAPI = {  
    .Open          = DRV_SST26_Open,  
    .Close         = DRV_SST26_Close,  
    .Status        = DRV_SST26_Status,  
    .SectorErase   = DRV_SST26_SectorErase,  
    .Read          = DRV_SST26_Read,  
    .PageWrite     = DRV_SST26_PageWrite,  
    .GeometryGet   = (DRV_MEMORY_DEVICE_GEOMETRY_GET)DRV_SST26_GeometryGet,  
    .TransferStatusGet = (DRV_MEMORY_DEVICE_TRANSFER_STATUS_GET)DRV_SST26_TransferStatusGet  
};  
  
const DRV_MEMORY_INIT drvMemory0InitData =  
{  
    .memDevIndex      = DRV_SST26_INDEX,  
    .memoryDevice     = &drvMemory0DeviceAPI,  
    .isFsEnabled       = true,  
    .deviceMediaType  = (uint8_t)SYS_FS_MEDIA_TYPE_SPIFLASH,  
    .ewBuffer          = &gDrvMemory0EraseBuffer[0],  
    .clientObjPool    = (uintptr_t)&gDrvMemory0ClientObject[0],  
    .bufferObj        = (uintptr_t)&gDrvMemory0BufferObject[0],  
    .queueSize         = DRV_MEMORY_BUFFER_QUEUE_SIZE_IDX0,  
    .nClientsMax      = DRV_MEMORY_CLIENTS_NUMBER_IDX0  
};  
  
//usage of DRV_MEMORY_INDEX_0 indicates usage of Flash-related APIs
```

```

objectHandle = DRV_MEMORY_Initialize((SYS_MODULE_INDEX)DRV_MEMORY_INDEX_0, (SYS_MODULE_INIT
*)&drvMemory0InitData);

if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}

```

Parameters

Parameters	Description
drvIndex	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_MEMORY_Initialize
(
const SYS_MODULE_INDEX drvIndex,
const SYS_MODULE_INIT * const init
);

```

DRV_MEMORY_Status Function

Gets the current status of the Memory driver module.

File

[drv_memory.h](#)

C

```
SYS_STATUS DRV_MEMORY_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.

SYS_STATUS_BUSY - Indicates the driver is in busy state.

Description

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

Remarks

This routine will NEVER block waiting for hardware.

Preconditions

Function [DRV_MEMORY_Initialize](#) should have been called before calling this function.

Example

```

SYS_MODULE_OBJ      object;      // Returned from DRV_MEMORY_Initialize
SYS_STATUS         MEMORYStatus;

MEMORYStatus = DRV_MEMORY_Status(object);

```

Parameters

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

Function

```

SYS_STATUS DRV_MEMORY_Status(SYS_MODULE_OBJ object);

```

DRV_MEMORY_Tasks Function

Maintains the Memory driver's internal state machine.

File

[drv_memory.h](#)

C

```
void DRV_MEMORY_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine maintains the driver's internal state machine.

Initial state is put to process Queue so that driver can accept transfer requests(open, erase, read, write, etc) from client.

This routine is also responsible for checking the status of Erase and Write transfer requests and notify the client through transferHandler registered if any.

The state of driver will be busy until transfer is completed. Once transfer is done the state goes back to Process state until new requests is received.

Remarks

This routine is generated only in Asynchronous mode.

This routine should not be called directly by application.

- For Bare Metal it will be called by the system Task routine (SYS_Tasks).
- For RTOS a separate Thread will be created for this task and will be called in the thread context.

Preconditions

The [DRV_MEMORY_Initialize](#) routine must have been called for the specified Memory driver instance.

Example

```
SYS_MODULE_OBJ object; // Returned from DRV_MEMORY_Initialize

void SYS_Tasks ( void )
{
    DRV_Memory_Tasks (object);
    // Do other tasks
}
```

Parameters

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

Function

```
void DRV_MEMORY_Tasks( SYS_MODULE_OBJ object );
```

b) Core Client Functions

DRV_MEMORY_Open Function

Opens the specified Memory driver instance and returns a handle to it

File

[drv_memory.h](#)

C

```
DRV_HANDLE DRV_MEMORY_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, [DRV_HANDLE_INVALID](#) is returned. Errors can occur under the following circumstances:

- if the attached media device is not ready or the geometry get fails.
- if the number of client objects allocated via [DRV_MEMORY_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
- if the attached memory device open or geometry read fails for first time.

Description

This routine opens the specified Memory driver instance and provides a handle.

When Open is called for the first time for specified Memory driver instance it opens the attached media device and reads the geometry details.

This handle returned 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_MEMORY_Close](#) routine is called.

This routine will NEVER block wait for hardware. If the driver has already been opened, it cannot be opened again.

Preconditions

Function [DRV_MEMORY_Initialize](#) must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_MEMORY_Open(DRV_MEMORY_INDEX_0);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the 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_MEMORY_Open
(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT ioIntent
);
```

DRV_MEMORY_Close Function

Closes an opened-instance of the Memory driver

File[drv_memory.h](#)**C**

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

Returns

None

Description

This routine closes an opened-instance of the Memory 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_MEMORY_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

[DRV_MEMORY_Open](#) must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle; // Returned from DRV_MEMORY_Open

DRV_MEMORY_Close(handle);
```

Parameters

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

Function

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

DRV_MEMORY_TransferHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

File[drv_memory.h](#)**C**

```
void DRV_MEMORY_TransferHandlerSet(const DRV_HANDLE handle, const void * transferHandler, const
uintptr_t context);
```

Returns

None.

Description

This function allows a client to set an event handling function for the driver to call back when queued operation has completed.

When a client calls a read, write, erase or a erasewrite function, it is provided with a handle identifying the command that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "transferHandler" function when the queued operation has completed.

The event handler should be set before the client performs any 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.
Used in Asynchronous Mode of operation.

Preconditions

The `DRV_MEMORY_Open()` routine must have been called to obtain a valid opened device handle.

Example

```
DRV_MEMORY_COMMAND_HANDLE commandHandle;

// memoryHandle is the handle returned by the DRV_MEMORY_Open function.
// Client registers an event handler with driver

// Event is received when the erase request is completed.
void appTransferHandler
(
    DRV_MEMORY_EVENT event,
    DRV_MEMORY_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    switch(event)
    {
        case DRV_MEMORY_EVENT_COMMAND_COMPLETE:
            xfer_done = true;
            break;
        case DRV_MEMORY_EVENT_COMMAND_ERROR:
            // Handle Error
            break;
        default:
            break;
    }
}

DRV_MEMORY_TransferHandlerSet(memoryHandle, appTransferHandler, (uintptr_t)NULL);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
transferHandler	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 transferHandler 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_MEMORY_TransferHandlerSet
(
    const DRV_HANDLE handle,
    const void * transferHandler,
    const uintptr_t context
);
```

c) Block Operation Functions

DRV_MEMORY_AsyncErase Function

Erase the specified number of memory blocks from the specified block start.

Each block is equal to sector size of the memory device attached.

File

[drv_memory.h](#)

C

```
void DRV_MEMORY_AsyncErase(const DRV_HANDLE handle, DRV_MEMORY_COMMAND_HANDLE * commandHandle,
                           uint32_t blockStart, uint32_t nBlock);
```

Returns

The command handle is returned in the commandHandle argument. It Will be [DRV_MEMORY_COMMAND_HANDLE_INVALID](#) if the request was not queued.

Description

This function schedules a non-blocking sector erase operation on attached memory device.

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_MEMORY_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 a transfer handler callback with the driver will issue a [DRV_MEMORY_EVENT_COMMAND_COMPLETE](#) event if the erase operation was successful or [DRV_MEMORY_EVENT_COMMAND_ERROR](#) event if the erase operation was not successful.

If the requesting client has not registered any transfer handler callback with the driver, he can call [DRV_MEMORY_CommandStatusGet\(\)](#) API to know the current status of the request.

Remarks

None.

Preconditions

The [DRV_MEMORY_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

```
// Use DRV_MEMORY_GeometryGet () to find the write region geometry.
uint32_t blockStart = 0;
uint32_t nBlocks = 10;
bool xfer_done = false;

// memoryHandle is the handle returned by the DRV_MEMORY_Open function.

// Event is received when the erase request is completed.
void appTransferHandler
(
    DRV_MEMORY_EVENT event,
    DRV_MEMORY_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    switch(event)
```

```

    {
        case DRV_MEMORY_EVENT_COMMAND_COMPLETE:
            xfer_done = true;
            break;
        case DRV_MEMORY_EVENT_COMMAND_ERROR:
            // Handle Error
            break;
        default:
            break;
    }
}

DRV_MEMORY_TransferHandlerSet(memoryHandle, appTransferHandler, (uintptr_t)NULL);

DRV_MEMORY_AsyncErase( memoryHandle, &commandHandle, blockStart, nBlock );

if(DRV_MEMORY_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Wait for erase to be completed
while(!xfer_done);

```

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	Block start from where the blocks should be erased.
nBlock	Total number of blocks to be erased.

Function

```

void DRV_MEMORY_AsyncErase
(
    const DRV_HANDLE handle,
    DRV_MEMORY_COMMAND_HANDLE *commandHandle,
    uint32_t blockStart,
    uint32_t nBlock
);

```

DRV_MEMORY_AsyncEraseWrite Function

Erase and Write blocks of data in the sectors where the block start belongs.

File

[drv_memory.h](#)

C

```

void DRV_MEMORY_AsyncEraseWrite(const DRV_HANDLE handle, DRV_MEMORY_COMMAND_HANDLE *
commandHandle, void * sourceBuffer, uint32_t blockStart, uint32_t nBlock);

```

Returns

The command handle is returned in the commandHandle argument. It Will be [DRV_MEMORY_COMMAND_HANDLE_INVALID](#) if the request was not queued.

Description

This function combines the step of erasing a sector and then writing the page. The application can use this function if it wants to avoid having to explicitly delete a sector in order to update the pages contained in the sector.

This function schedules a non-blocking operation to erase and write blocks of data into attached device memory.

The function returns with a valid command 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_MEMORY_COMMAND_HANDLE_INVALID](#) in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the sourceBuffer 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 driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a [DRV_MEMORY_EVENT_COMMAND_COMPLETE](#) event if the buffer was processed successfully or [DRV_MEMORY_EVENT_COMMAND_ERROR](#) event if the buffer was not processed successfully.

If the requesting client has not registered any transfer handler callback with the driver, he can call [DRV_MEMORY_CommandStatusGet\(\)](#) API to know the current status of the request.

Remarks

None.

Preconditions

The [DRV_MEMORY_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

```
#define BUFFER_SIZE      4096
uint8_t buffer[BUFFER_SIZE];

// Use DRV_MEMORY_GeometryGet () to find the write region geometry.
uint32_t blockStart = 0x0;
uint32_t nBlock = BUFFER_SIZE / block_size; // block_size for write geometry
DRV_MEMORY_COMMAND_HANDLE commandHandle;

// memoryHandle is the handle returned by the DRV_MEMORY_Open function.
// Client registers an event handler with driver

// Event is received when the erase request is completed.
void appTransferHandler
(
    DRV_MEMORY_EVENT event,
    DRV_MEMORY_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    switch(event)
    {
        case DRV_MEMORY_EVENT_COMMAND_COMPLETE:
            xfer_done = true;
            break;
        case DRV_MEMORY_EVENT_COMMAND_ERROR:
            // Handle Error
            break;
        default:
            break;
    }
}

DRV_MEMORY_TransferHandlerSet(memoryHandle, appTransferHandler, (uintptr_t)NULL);

DRV_MEMORY_AsyncEraseWrite(memoryHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_MEMORY_COMMAND_HANDLE_INVALID == commandHandle)
```

```

{
    // Error handling here
}

// Wait for erase to be completed
while(!xfer_done);

```

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 command handle. If NULL, then command handle is not returned.
sourceBuffer	The source buffer containing data to be programmed into media device memory
blockStart	Write block start where the write should begin.
nBlock	Total number of blocks to be written.

Function

```

void DRV_MEMORY_AsyncEraseWrite
(
    const DRV_HANDLE handle,
    DRV_MEMORY_COMMAND_HANDLE * commandHandle,
    void * sourceBuffer,
    uint32_t blockStart,
    uint32_t nBlock
);

```

DRV_MEMORY_AsyncRead Function

Reads nblocks of data from the specified block start.

File

[drv_memory.h](#)

C

```

void DRV_MEMORY_AsyncRead(const DRV_HANDLE handle, DRV_MEMORY_COMMAND_HANDLE * commandHandle,
    void * targetBuffer, uint32_t blockStart, uint32_t nBlock);

```

Returns

The command handle is returned in the commandHandle argument. It will be [DRV_MEMORY_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 memory device attached.

The function returns with a valid command handle in the commandHandle argument if the 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_MEMORY_COMMAND_HANDLE_INVALID](#) in the commandHandle argument under the following circumstances:

- if a buffer object 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 number of blocks to be read is either zero or more than the number of blocks actually available
- if the driver handle is invalid

Remarks

None.

Preconditions

`DRV_MEMORY_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 readBuffer[BUFFER_SIZE];

// Use DRV_MEMORY_GeometryGet () to find the read region geometry.
// Find the block address from which to read data.
uint32_t blockStart = 0x0;
uint32_t nBlock = BUFFER_SIZE;
DRV_MEMORY_COMMAND_HANDLE commandHandle;
bool xfer_done = false;

// memoryHandle is the handle returned by the DRV_MEMORY_Open function.

// Event is received when the write request is completed.
void appTransferHandler
(
    DRV_MEMORY_EVENT event,
    DRV_MEMORY_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    switch(event)
    {
        case DRV_MEMORY_EVENT_COMMAND_COMPLETE:
            xfer_done = true;
            break;
        case DRV_MEMORY_EVENT_COMMAND_ERROR:
            // Handle Error
            break;
        default:
            break;
    }
}

DRV_MEMORY_TransferHandlerSet(memoryHandle, appTransferHandler, (uintptr_t)NULL);

DRV_MEMORY_AsyncRead(memoryHandle, &commandHandle, &readBuffer, blockStart, nBlock);

if(DRV_MEMORY_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

while(!xfer_done);
```

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 command handle
targetBuffer	Buffer into which the data read from the media device memory will be placed
blockStart	Block start from where the data should be read.
nBlock	Total number of blocks to be read.

Function

`void DRV_MEMORY_AsyncRead`

```

(
const    DRV_HANDLE handle,
DRV_MEMORY_COMMAND_HANDLE *commandHandle,
void *targetBuffer,
uint32_t blockStart,
uint32_t nBlock
);

```

DRV_MEMORY_AsyncWrite Function

Writes nblocks of data starting at the specified block start.

File

[drv_memory.h](#)

C

```

void DRV_MEMORY_AsyncWrite(const DRV_HANDLE handle, DRV_MEMORY_COMMAND_HANDLE * commandHandle,
void * sourceBuffer, uint32_t blockStart, uint32_t nBlock);

```

Returns

The command handle is returned in the commandHandle argument. It will be [DRV_MEMORY_COMMAND_HANDLE_INVALID](#) if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into attached devices memory.

The function returns with a valid command 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_MEMORY_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 driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a [DRV_MEMORY_EVENT_COMMAND_COMPLETE](#) event if the buffer was processed successfully or [DRV_MEMORY_EVENT_COMMAND_ERROR](#) event if the buffer was not processed successfully.

If the requesting client has not registered any transfer handler callback with the driver, he can call [DRV_MEMORY_CommandStatusGetGet\(\)](#) API to know the current status of the request.

Remarks

None.

Preconditions

[DRV_MEMORY_Open\(\)](#) routine must have been called to obtain a valid opened device handle.

The memory address location which has to be written, must have been erased before using the [DRV_MEMORY_xxxErase\(\)](#) routine.

Example

```

#define BUFFER_SIZE    4096
uint8_t writeBuffer[BUFFER_SIZE];

// Use DRV_MEMORY_GeometryGet () to find the write region geometry.

```

```

uint32_t blockStart = 0x0;
uint32_t nBlock = BUFFER_SIZE / block_size; // block_size for write geometry
bool xfer_done = false;

DRV_MEMORY_COMMAND_HANDLE commandHandle;

// memoryHandle is the handle returned by the DRV_MEMORY_Open function.

// Event is received when the write request is completed.
void appTransferHandler
(
    DRV_MEMORY_EVENT event,
    DRV_MEMORY_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    switch(event)
    {
        case DRV_MEMORY_EVENT_COMMAND_COMPLETE:
            xfer_done = true;
            break;
        case DRV_MEMORY_EVENT_COMMAND_ERROR:
            // Handle Error
            break;
        default:
            break;
    }
}

DRV_MEMORY_TransferHandlerSet(memoryHandle, appTransferHandler, (uintptr_t)NULL);

DRV_MEMORY_AsyncErase(memoryHandle, &commandHandle, blockStart, nBlock);

if(DRV_MEMORY_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Wait for erase to be completed
while(!xfer_done);

DRV_MEMORY_AsyncWrite(memoryHandle, &commandHandle, &writeBuffer, blockStart, nBlock);

if(DRV_MEMORY_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Wait for write to be completed
while(!xfer_done);

```

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 media device memory
blockStart	Block start from where the data should be written to.
nBlock	Total number of blocks to be written.

Function

```

void DRV_MEMORY_AsyncWrite
(
    const DRV_HANDLE handle,

```

```
DRV_MEMORY_COMMAND_HANDLE *commandHandle,
void *sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
);
```

DRV_MEMORY_SyncErase Function

Erase the specified number of memory blocks from the specified block start.

Each block is equal to sector size of the memory device attached.

File

[drv_memory.h](#)

C

```
bool DRV_MEMORY_SyncErase(const DRV_HANDLE handle, uint32_t blockStart, uint32_t nBlock);
```

Returns

true:

- If the transfer request is successfully completed.

false:

- 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 driver handle is invalid

Description

This function schedules a blocking sector erase operation on attached memory device.

Remarks

None.

Preconditions

The [DRV_MEMORY_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

```
// Use DRV_MEMORY_GeometryGet() to find the erase region geometry.
uint32_t blockStart = 0;
uint32_t nBlocks = 10;

// memoryHandle is the handle returned by the DRV_MEMORY_Open function.

if(DRV_MEMORY_SyncErase( memoryHandle, blockStart, nBlock ) == false)
{
    // Error handling here
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
blockStart	block start from where the blocks should be erased.
nBlock	Total number of blocks to be erased.

Function

```
bool DRV_MEMORY_SyncErase
()
```

```
const     DRV_HANDLE handle,
uint32_t blockStart,
uint32_t nBlock
);
```

DRV_MEMORY_Erase Function

This routine provides interface to the file system to perform a media erase operation in synchronous mode of the Memory driver.

File

[drv_memory.h](#)

C

```
void DRV_MEMORY_Erase(const DRV_HANDLE handle, SYS_MEDIA_BLOCK_COMMAND_HANDLE * commandHandle,
uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be [DRV_MEMORY_COMMAND_HANDLE_INVALID](#) if the request was not successful.

Description

This function is internally used by the file system.

Remarks

This function is internally used by the file system.

Preconditions

The [DRV_MEMORY_Open](#) must have been called to obtain a valid opened device handle.

Example

None.

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 of SD Card where the writes should begin.
nBlock	Total number of blocks to be written.

Function

```
void DRV_MEMORY_Erase
(
const     DRV_HANDLE handle,
SYS_MEDIA_BLOCK_COMMAND_HANDLE* commandHandle,
void* sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
)
```

DRV_MEMORY_SyncEraseWrite Function

Erase and Write blocks of data in the sectors where the block start belongs.

File

[drv_memory.h](#)

C

```
bool DRV_MEMORY_SyncEraseWrite(const DRV_HANDLE handle, void * sourceBuffer, uint32_t
blockStart, uint32_t nBlock);
```

Returns

true:

- If the transfer request is successfully completed.

false:

- If the sourceBuffer 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 driver handle is invalid

Description

This function combines the step of erasing a sector and then writing the page. The application can use this function if it wants to avoid having to explicitly delete a sector in order to update the pages contained in the sector.

This function schedules a blocking operation to erase and write blocks of data into attached device memory.

Remarks

None.

Preconditions

The [DRV_MEMORY_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

```
#define BUFFER_SIZE 4096

uint8_t buffer[BUFFER_SIZE];

// Use DRV_MEMORY_GeometryGet () to find the write region geometry.
uint32_t blockStart = 0x0;
uint32_t nBlock = BUFFER_SIZE / block_size; // block_size for write geometry

// memoryHandle is the handle returned by the DRV_MEMORY_Open function.

if(DRV_MEMORY_SyncEraseWrite(memoryHandle, &myBuffer, blockStart, nBlock) == false)
{
    // Error handling here
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
sourceBuffer	The source buffer containing data to be programmed into media device memory
blockStart	block start where the write should begin.
nBlock	Total number of blocks to be written.

Function

```
bool DRV_MEMORY_SyncEraseWrite
(
const DRV_HANDLE handle,
void *sourceBuffer,
uint32_t blockStart,
```

```
    uint32_t nBlock
);
```

DRV_MEMORY_EraseWrite Function

This routine provides interface to the file system to perform a media erase-write operation in synchronous mode of the Memory driver.

File

[drv_memory.h](#)

C

```
void DRV_MEMORY_EraseWrite(const DRV_HANDLE handle, SYS_MEDIA_BLOCK_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_MEMORY_COMMAND_HANDLE_INVALID](#) if the request was not successful.

Description

This function is internally used by the file system.

Remarks

This function is internally used by the file system.

Preconditions

The [DRV_MEMORY_Open](#) must have been called to obtain a valid opened device handle.

Example

None.

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_MEMORY_EraseWrite  
(  
    const DRV_HANDLE handle,  
    SYS_MEDIA_BLOCK_COMMAND_HANDLE* commandHandle,  
    void* sourceBuffer,  
    uint32_t blockStart,  
    uint32_t nBlock  
)
```

DRV_MEMORY_SyncRead Function

Reads nblock of data from the specified block start.

File

[drv_memory.h](#)

C

```
bool DRV_MEMORY_SyncRead(const DRV_HANDLE handle, void * targetBuffer, uint32_t blockStart,  
uint32_t nBlock);
```

Returns

true:

- If the transfer request is successfully completed.

false:

- If the target buffer pointer is NULL
- If the client opened the driver for write only
- If the number of blocks to be read is either zero or more than the number of blocks actually available
- If the driver handle is invalid

Description

This function schedules a blocking read operation for reading blocks of data from the memory device attached.

Remarks

None.

Preconditions

`DRV_MEMORY_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

```
#define BUFFER_SIZE 4096  
uint8_t readBuffer[BUFFER_SIZE];  
  
// Use DRV_MEMORY_GeometryGet () to find the read region geometry.  
uint32_t blockStart = 0x0;  
uint32_t nBlock = BUFFER_SIZE / block_size; // block_size for read geometry  
  
// memoryHandle is the handle returned by the DRV_MEMORY_Open function.  
  
if(DRV_MEMORY_SyncRead(memoryHandle, &readBuffer, blockStart, nBlock) == false)  
{  
    // Error handling here  
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
targetBuffer	Buffer into which the data read from the media device memory will be placed
blockStart	Block start from where the data should be read.
nBlock	Total number of blocks to be read.

Function

```
bool DRV_MEMORY_SyncRead  
(  
    const DRV_HANDLE handle,  
    void *targetBuffer,  
    uint32_t blockStart,  
    uint32_t nBlock  
>;
```

DRV_MEMORY_Read Function

This routine provides interface to the file system to perform a media read operation in synchronous mode of the Memory driver.

File

[drv_memory.h](#)

C

```
void DRV_MEMORY_Read(const DRV_HANDLE handle, SYS_MEDIA_BLOCK_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_MEMORY_COMMAND_HANDLE_INVALID](#) if the request was not successful.

Description

This function is internally used by the file system.

Remarks

This function is internally used by the file system.

Preconditions

The [DRV_MEMORY_Open](#) must have been called to obtain a valid opened device handle.

Example

None.

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_MEMORY_Read
(
const DRV_HANDLE handle,
SYS_MEDIA_BLOCK_COMMAND_HANDLE* commandHandle,
void* sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
)
```

DRV_MEMORY_SyncWrite Function

Writes nblock of data starting at the specified block start.

File

[drv_memory.h](#)

C

```
bool DRV_MEMORY_SyncWrite(const DRV_HANDLE handle, void * sourceBuffer, uint32_t blockStart,
                           uint32_t nBlock);
```

Returns

true:

- If the transfer request is successfully completed.

false:

- 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 driver handle is invalid

Description

This function schedules a blocking write operation for writing blocks of data into attached devices memory.

Remarks

None.

Preconditions

DRV_MEMORY_Open() routine must have been called to obtain a valid opened device handle.

The memory block which has to be written, must have been erased before using the DRV_MEMORY_xxxErase() routine.

Example

```
#define BUFFER_SIZE 4096
uint8_t writeBuffer[BUFFER_SIZE];

// Use DRV_MEMORY_GeometryGet () to find the write region geometry.
uint32_t blockStart = 0x0;
uint32_t nBlock = BUFFER_SIZE / block_size; // block_size for write geometry

// memoryHandle is the handle returned by the DRV_MEMORY_Open function.

if(DRV_MEMORY_SyncErase(memoryHandle, blockStart, nBlock) == false)
{
    // Error handling here
}

if(DRV_MEMORY_SyncWrite(memoryHandle, &writeBuffer, blockStart, nBlock) == false)
{
    // Error handling here
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
sourceBuffer	The source buffer containing data to be programmed into media device memory
blockStart	Block start from where the data should be written to.
nBlock	Total number of blocks to be written.

Function

```
bool DRV_MEMORY_SyncWrite
(
    const DRV_HANDLE handle,
    void *sourceBuffer,
    uint32_t blockStart,
    uint32_t nBlock
```

);

DRV_MEMORY_Write Function

This routine provides interface to the file system to perform a media write operation in synchronous mode of the Memory driver.

File

[drv_memory.h](#)

C

```
void DRV_MEMORY_Write(const DRV_HANDLE handle, SYS_MEDIA_BLOCK_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_MEMORY_COMMAND_HANDLE_INVALID](#) if the request was not successful.

Description

This function is internally used by the file system.

Remarks

This function is internally used by the file system.

Preconditions

The [DRV_MEMORY_Open](#) must have been called to obtain a valid opened device handle.

Example

None.

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_MEMORY_Write  
(  
    const DRV_HANDLE handle,  
    SYS_MEDIA_BLOCK_COMMAND_HANDLE* commandHandle,  
    void* sourceBuffer,  
    uint32_t blockStart,  
    uint32_t nBlock  
)
```

DRV_MEMORY_CommandStatusGet Function

Gets the current status of the command.

File

[drv_memory.h](#)

C

```
DRV_MEMORY_COMMAND_STATUS DRV_MEMORY_CommandStatusGet(const DRV_HANDLE handle, const
DRV_MEMORY_COMMAND_HANDLE commandHandle);
```

Returns

DRV_MEMORY_COMMAND_COMPLETED

- If the transfer request is completed

DRV_MEMORY_COMMAND_QUEUED

- If the command is Queued and waiting to be processed.

DRV_MEMORY_COMMAND_IN_PROGRESS

- If the current transfer request is still being processed

DRV_MEMORY_COMMAND_ERROR_UNKNOWN

- If the handle is invalid

- If the status read request fails

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_MEMORY_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 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 or erase operation completion events.

Remarks

Used in Async mode of operation.

Preconditions

DRV_MEMORY_Open() must have been called to obtain a valid opened device handle.

Example

```
// memoryHandle is the handle returned by the DRV_MEMORY_Open function.
// commandHandle is the handle returned by any read/write/erase block operation.

if (DRV_MEMORY_COMMAND_COMPLETED == DRV_MEMORY_CommandStatusGet(memoryHandle, commandHandle))
{
    // Operation Done
}
```

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 command handle

Function

```
DRV_MEMORY_COMMAND_STATUS DRV_MEMORY_CommandStatusGet
(
const DRV_HANDLE handle,
const DRV_MEMORY_COMMAND_HANDLE commandHandle
);
```

DRV_MEMORY_TransferStatusGet Function

Gets the current status of the transfer request on attached device.

File

[drv_memory.h](#)

C

```
MEMORY_DEVICE_TRANSFER_STATUS DRV_MEMORY_TransferStatusGet(const DRV_HANDLE handle);
```

Returns

MEMORY_DEVICE_TRANSFER_ERROR_UNKNOWN

- If the handle is invalid
- If the status read request fails

MEMORY_DEVICE_TRANSFER_BUSY

- If the current transfer request is still being processed

MEMORY_DEVICE_TRANSFER_COMPLETED

- If the transfer request is completed

Description

This routine gets the current status of the transfer request. The application must use this routine where the status of a scheduled request needs to be polled on.

The application can alternatively register a transfer handler to receive the transfer completion events.

Remarks

This routine will block for hardware access.

Used in Async mode of operation.

Preconditions

[DRV_MEMORY_Open\(\)](#) must have been called to obtain a valid opened device handle.

Example

```
// memoryHandle is the handle returned by the DRV_MEMORY_Open function.

if (MEMORY_DEVICE_TRANSFER_COMPLETED == DRV_MEMORY_TransferStatusGet(memoryHandle))
{
    // Operation Done
}
```

Parameters

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

Function

```
MEMORY_DEVICE_TRANSFER_STATUS DRV_MEMORY_TransferStatusGet
(
const DRV_HANDLE handle
);
```

d) Media Interface Functions

DRV_MEMORY_AddressGet Function

Returns the Memory media start address

File[drv_memory.h](#)**C**

```
uintptr_t DRV_MEMORY_AddressGet(const DRV_HANDLE handle);
```

Returns

Start address of the Memory Media if the handle is valid otherwise NULL.

Description

This function returns the Memory Media start address.

Remarks

None.

Preconditions

The [DRV_MEMORY_Open\(\)](#) routine must have been called to obtain a valid opened device handle.

Example

```
uintptr_t startAddress;
startAddress = DRV_MEMORY_AddressGet(drvMEMORYHandle);
```

Parameters

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

Function

```
uintptr_t DRV_MEMORY_AddressGet
(
const DRV_HANDLE handle
);
```

DRV_MEMORY_GeometryGet Function

Returns the geometry of the memory device.

File[drv_memory.h](#)**C**

```
SYS_MEDIA_GEOMETRY * DRV_MEMORY_GeometryGet(const DRV_HANDLE handle);
```

Returns

[SYS_MEDIA_GEOMETRY](#) - Pointer to structure which holds the media geometry information.

Description

This API gives the following geometrical details of the attached memory device:

- Media Property
- Number of Read/Write/Erase regions in the memory device
- Number of Blocks and their size in each region of the device

Remarks

Refer sys_media.h for definition of [SYS_MEDIA_GEOMETRY](#).

Preconditions

The [DRV_MEMORY_Open\(\)](#) routine must have been called to obtain a valid opened device handle.

Example

```

SYS_MEDIA_GEOMETRY geometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

if (true != DRV_MEMORY_GeometryGet(&geometry))
{
    // Handle Error
}

readBlockSize = geometry.geometryTable[0].blockSize;
nReadBlocks = geometry.geometryTable[0].numBlocks;
nReadRegions = geometry.numReadRegions;

writeBlockSize = geometry.geometryTable[1].blockSize;
eraseBlockSize = geometry.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_MEDIA_GEOMETRY * DRV_MEMORY_GeometryGet
(
const DRV_HANDLE handle
);

```

DRV_MEMORY_IsAttached Function

Returns the physical attach status of the Media.

File

[drv_memory.h](#)

C

```
bool DRV_MEMORY_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 Media device.

Remarks

None.

Preconditions

The [DRV_MEMORY_Open\(\)](#) routine must have been called to obtain a valid opened device handle.

Example

```

bool isMEMORYAttached;
isMEMORYAttached = DRV_MEMORY_isAttached(drvMEMORYHandle);

```

Parameters

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

Function

```
bool DRV_MEMORY_IsAttached
(
const DRV_HANDLE handle
);
```

DRV_MEMORY_IsWriteProtected Function

Returns the write protect status of the Memory.

File

[drv_memory.h](#)

C

```
bool DRV_MEMORY_IsWriteProtected(const DRV_HANDLE handle);
```

Returns

True - If the memory is write protected. False - If the memory is not write protected.

Description

This function returns the write protect status of the Memory.

Remarks

None.

Preconditions

The [DRV_MEMORY_Open\(\)](#) routine must have been called to obtain a valid opened device handle.

Example

```
bool isWriteProtected;
isWriteProtected = DRV_MEMORY_IsWriteProtected(drvMEMORYHandle);
```

Parameters

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

Function

```
bool DRV_MEMORY_IsWriteProtected
(
const DRV_HANDLE handle
);
```

e) Data Types and Constants

DRV_MEMORY_COMMAND_HANDLE Type

Handle to identify commands queued in the driver.

File[drv_memory.h](#)**C**

```
typedef SYS_MEDIA_BLOCK_COMMAND_HANDLE DRV_MEMORY_COMMAND_HANDLE;
```

Description

A command handle is returned by a call to the Read, Write, Erase or EraseWrite 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

Refer sys_media.h for definition of [SYS_MEDIA_BLOCK_COMMAND_HANDLE](#).

DRV_MEMORY_COMMAND_HANDLE_INVALID Macro

This value defines the Memory Driver's Invalid Command Handle.

File[drv_memory.h](#)**C**

```
#define DRV_MEMORY_COMMAND_HANDLE_INVALID SYS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID
```

Description

This value defines the Memory Driver's Invalid Command Handle. This value is returned by read/write/erase/erasure routines when the command request was not accepted.

Remarks

Refer sys_media.h for definition of [SYS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID](#).

DRV_MEMORY_COMMAND_STATUS Enumeration

Memory Driver command Status

File[drv_memory.h](#)**C**

```
typedef enum {
    DRV_MEMORY_COMMAND_COMPLETED = SYS_MEDIA_COMMAND_COMPLETED,
    DRV_MEMORY_COMMAND_QUEUED = SYS_MEDIA_COMMAND_QUEUED,
    DRV_MEMORY_COMMAND_IN_PROGRESS = SYS_MEDIA_COMMAND_IN_PROGRESS,
    DRV_MEMORY_COMMAND_ERROR_UNKNOWN = SYS_MEDIA_COMMAND_UNKNOWN
} DRV_MEMORY_COMMAND_STATUS;
```

Members

Members	Description
DRV_MEMORY_COMMAND_COMPLETED = SYS_MEDIA_COMMAND_COMPLETED	Done OK and ready

DRV_MEMORY_COMMAND_QUEUED = SYS_MEDIA_COMMAND_QUEUED	Scheduled but not started
DRV_MEMORY_COMMAND_IN_PROGRESS = SYS_MEDIA_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_MEMORY_COMMAND_ERROR_UNKNOWN = SYS_MEDIA_COMMAND_UNKNOWN	Unknown Command

Description

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

Remarks

Refer sys_media.h for SYS_MEDIA_XXX definitions.

DRV_MEMORY_DEVICE_INTERFACE Structure

Memory Device API.

File

[drv_memory_definitions.h](#)

C

```
typedef struct {
    DRV_MEMORY_DEVICE_OPEN Open;
    DRV_MEMORY_DEVICE_CLOSE Close;
    DRV_MEMORY_DEVICE_SECTOR_ERASE SectorErase;
    DRV_MEMORY_DEVICE_STATUS Status;
    DRV_MEMORY_DEVICE_READ Read;
    DRV_MEMORY_DEVICE_PAGE_WRITE PageWrite;
    DRV_MEMORY_DEVICE_EVENT_HANDLER_SET EventHandlerSet;
    DRV_MEMORY_DEVICE_GEOMETRY_GET GeometryGet;
    DRV_MEMORY_DEVICE_TRANSFER_STATUS_GET TransferStatusGet;
} DRV_MEMORY_DEVICE_INTERFACE;
```

Description

This Data Structure is used by attached media to populate the required device functions for media transactions.

This will be used in memory driver init structure.

Remarks

None.

DRV_MEMORY_DEVICE_OPEN Type

File

[drv_memory_definitions.h](#)

C

```
typedef DRV_HANDLE (* DRV_MEMORY_DEVICE_OPEN)(const SYS_MODULE_INDEX drvIndex, const
DRV_IO_INTENT ioIntent);
```

Description

Function pointer typedef to open the attached media

DRV_MEMORY_DEVICE_CLOSE Type

File

[drv_memory_definitions.h](#)

C

```
typedef void (* DRV_MEMORY_DEVICE_CLOSE)(const DRV_HANDLE handle);
```

Description

Function pointer typedef to close the attached media

DRV_MEMORY_DEVICE_SECTOR_ERASE Type

File

[drv_memory_definitions.h](#)

C

```
typedef bool (* DRV_MEMORY_DEVICE_SECTOR_ERASE)(const DRV_HANDLE handle, uint32_t address);
```

Description

Function pointer typedef to erase a sector from attached media

DRV_MEMORY_DEVICE_STATUS Type

File

[drv_memory_definitions.h](#)

C

```
typedef SYS_STATUS (* DRV_MEMORY_DEVICE_STATUS)(const SYS_MODULE_INDEX drvIndex);
```

Description

Function pointer typedef to get the status of the attached media

DRV_MEMORY_DEVICE_READ Type

File

[drv_memory_definitions.h](#)

C

```
typedef bool (* DRV_MEMORY_DEVICE_READ)(const DRV_HANDLE handle, void *rx_data, uint32_t rx_data_length, uint32_t address);
```

Description

Function pointer typedef to read from the attached media

DRV_MEMORY_DEVICE_PAGE_WRITE Type

File

[drv_memory_definitions.h](#)

C

```
typedef bool (* DRV_MEMORY_DEVICE_PAGE_WRITE)(const DRV_HANDLE handle, void *tx_data, uint32_t address);
```

Description

Function pointer typedef to write a page to the attached media

DRV_MEMORY_DEVICE_EVENT_HANDLER_SET Type**File**

[drv_memory_definitions.h](#)

C

```
typedef void (* DRV_MEMORY_DEVICE_EVENT_HANDLER_SET)(const DRV_HANDLE handle, DRV_MEMORY_EVENT_HANDLER eventHandler, uintptr_t context);
```

Description

Function pointer typedef to set the event handler with attached media

DRV_MEMORY_DEVICE_GEOMETRY_GET Type**File**

[drv_memory_definitions.h](#)

C

```
typedef bool (* DRV_MEMORY_DEVICE_GEOMETRY_GET)(const DRV_HANDLE handle, MEMORY_DEVICE_GEOMETRY *geometry);
```

Description

Function pointer typedef to get the Geometry details from attached media

DRV_MEMORY_DEVICE_TRANSFER_STATUS_GET Type**File**

[drv_memory_definitions.h](#)

C

```
typedef uint32_t (* DRV_MEMORY_DEVICE_TRANSFER_STATUS_GET)(const DRV_HANDLE handle);
```

Description

Function pointer typedef to get the transfer Status from attached media

DRV_MEMORY_EVENT Enumeration

Identifies the possible events that can result from a request.

File

[drv_memory.h](#)

C

```
typedef enum {
```

```

DRV_MEMORY_EVENT_COMMAND_COMPLETE = SYS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE,
DRV_MEMORY_EVENT_COMMAND_ERROR = SYS_MEDIA_EVENT_BLOCK_COMMAND_ERROR
} DRV_MEMORY_EVENT;

```

Members

Members	Description
DRV_MEMORY_EVENT_COMMAND_COMPLETE = SYS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE	Operation has been completed successfully.
DRV_MEMORY_EVENT_COMMAND_ERROR = SYS_MEDIA_EVENT_BLOCK_COMMAND_ERROR	There was an error during the operation

Description

This enumeration identifies the possible events that can result from a read, write, erase or erasewrite 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_MEMORY_TransferHandlerSet](#) function when a request is completed.

Refer [sys_media.h](#) for SYS_MEDIA_XXX definitions.

DRV_MEMORY_EVENT_HANDLER Type

File

[drv_memory_definitions.h](#)

C

```
typedef void (* DRV_MEMORY_EVENT_HANDLER)(MEMORY_DEVICE_TRANSFER_STATUS status, uintptr_t context);
```

Description

Function pointer typedef for event handler to be sent to attached media

DRV_MEMORY_INIT Structure

Memory Driver Initialization Data

File

[drv_memory_definitions.h](#)

C

```

typedef struct {
    SYS_MODULE_INDEX memDevIndex;
    const DRV_MEMORY_DEVICE_INTERFACE * memoryDevice;
    bool isMemDevInterruptEnabled;
    uint32_t memDevStatusPollUs;
    bool isFsEnabled;
    uint8_t deviceMediaType;
    uint8_t * ewBuffer;
    uintptr_t clientObjPool;
    uintptr_t bufferObj;
    size_t queueSize;
    size_t nClientsMax;
} DRV_MEMORY_INIT;

```

Members

Members	Description
SYS_MODULE_INDEX memDevIndex;	Attached Memory Device index

const DRV_MEMORY_DEVICE_INTERFACE * memoryDevice;	Flash Device functions
bool isMemDevInterruptEnabled;	Flag to indicate if attached memory device configured to interrupt mode
uint32_t memDevStatusPollUs;	Number of milliseconds to poll for transfer status check
bool isFsEnabled;	FS enabled
uint8_t deviceMediaType;	Memory Device Type
uint8_t * ewBuffer;	Erase Write Buffer pointer
uintptr_t clientObjPool;	Memory pool for Client Objects
uintptr_t bufferObj;	Pointer to Buffer Objects array
size_t queueSize;	Buffer Queue Size
size_t nClientsMax;	Maximum number of clients

Description

This data type defines the data required to initialize the Memory Driver.

Remarks

Not all initialization features are available for all devices. Please refer to the attached media device capabilities.

DRV_MEMORY_TRANSFER_HANDLER Type

Pointer to a Memory Driver Event handler function

File

[drv_memory.h](#)

C

```
typedef SYS_MEDIA_EVENT_HANDLER DRV_MEMORY_TRANSFER_HANDLER;
```

Returns

None.

Description

This data type defines the required function signature for the Memory 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_MEMORY_EVENT_COMMAND_COMPLETE, it means that the requested operation was completed successfully.

If the event is DRV_MEMORY_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_MEMORY_TransferHandlerSet](#) 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.

Used in Asynchronous mode of operation.

Refer [sys_media.h](#) for definition of [SYS_MEDIA_EVENT_HANDLER](#).

Example

```
void appTransferHandler
(
    DRV_MEMORY_EVENT event,
    DRV_MEMORY_COMMAND_HANDLE commandHandle,
```

```

        uintptr_t context
    )
{
    switch(event)
    {
        case DRV_MEMORY_EVENT_COMMAND_COMPLETE:
            xfer_done = true;
            break;
        case DRV_MEMORY_EVENT_COMMAND_ERROR:
            // Handle Error
            break;
        default:
            break;
    }
}

```

Parameters

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

MEMORY_DEVICE_GEOMETRY Structure

Memory Device Geometry Table.

File

[drv_memory_definitions.h](#)

C

```

typedef struct {
    uint32_t read_blockSize;
    uint32_t read_numBlocks;
    uint32_t numReadRegions;
    uint32_t write_blockSize;
    uint32_t write_numBlocks;
    uint32_t numWriteRegions;
    uint32_t erase_blockSize;
    uint32_t erase_numBlocks;
    uint32_t numEraseRegions;
    uint32_t blockStartAddress;
} MEMORY_DEVICE_GEOMETRY;

```

Description

This Data Structure is used by Memory driver to get the media geometry details.

The Media attached to memory driver needs to fill in this data structure when GEOMETRY_GET is called.

Remarks

None.

MEMORY_DEVICE_TRANSFER_STATUS Enumeration

Memory Device Transfer Status.

File

[drv_memory_definitions.h](#)

C

```
typedef enum {
    MEMORY_DEVICE_TRANSFER_BUSY,
    MEMORY_DEVICE_TRANSFER_COMPLETED,
    MEMORY_DEVICE_TRANSFER_ERROR_UNKNOWN
} MEMORY_DEVICE_TRANSFER_STATUS;
```

Members

Members	Description
MEMORY_DEVICE_TRANSFER_BUSY	Transfer being processed
MEMORY_DEVICE_TRANSFER_COMPLETED	Transfer is successfully completed
MEMORY_DEVICE_TRANSFER_ERROR_UNKNOWN	Transfer had error

Description

This Data structure is used to indicate the current transfer status of the attached media.

Remarks

None.

Files**Files**

Name	Description
drv_memory.h	Memory Driver Interface Definition
drv_memory_definitions.h	Memory Driver Interface Definition

Description

This section will list only the library's interface header file(s).

drv_memory.h

Memory Driver Interface Definition

Enumerations

	Name	Description
	DRV_MEMORY_COMMAND_STATUS	Memory Driver command Status
	DRV_MEMORY_EVENT	Identifies the possible events that can result from a request.

Functions

	Name	Description
≡	DRV_MEMORY_AddressGet	Returns the Memory media start address
≡	DRV_MEMORY_AsyncErase	Erase the specified number of memory blocks from the specified block start. Each block is equal to sector size of the memory device attached.
≡	DRV_MEMORY_AsyncEraseWrite	Erase and Write blocks of data in the sectors where the block start belongs.
≡	DRV_MEMORY_AsyncRead	Reads nblocks of data from the specified block start.
≡	DRV_MEMORY_AsyncWrite	Writes nblocks of data starting at the specified block start.
≡	DRV_MEMORY_Close	Closes an opened-instance of the Memory driver
≡	DRV_MEMORY_CommandStatusGet	Gets the current status of the command.
≡	DRV_MEMORY_Erase	This routine provides interface to the file system to perform a media erase operation in synchronous mode of the Memory driver.

DRV_MEMORY_EraseWrite	This routine provides interface to the file system to perform a media erase-write operation in synchronous mode of the Memory driver.
DRV_MEMORY_GeometryGet	Returns the geometry of the memory device.
DRV_MEMORY_Initialize	Initializes the Memory instance for the specified driver index
DRV_MEMORY_IsAttached	Returns the physical attach status of the Media.
DRV_MEMORY_IsWriteProtected	Returns the write protect status of the Memory.
DRV_MEMORY_Open	Opens the specified Memory driver instance and returns a handle to it
DRV_MEMORY_Read	This routine provides interface to the file system to perform a media read operation in synchronous mode of the Memory driver.
DRV_MEMORY_Status	Gets the current status of the Memory driver module.
DRV_MEMORY_SyncErase	Erase the specified number of memory blocks from the specified block start. Each block is equal to sector size of the memory device attached.
DRV_MEMORY_SyncEraseWrite	Erase and Write blocks of data in the sectors where the block start belongs.
DRV_MEMORY_SyncRead	Reads nblock of data from the specified block start.
DRV_MEMORY_SyncWrite	Writes nblock of data starting at the specified block start.
DRV_MEMORY_Tasks	Maintains the Memory driver's internal state machine.
DRV_MEMORY_TransferHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.
DRV_MEMORY_TransferStatusGet	Gets the current status of the transfer request on attached device.
DRV_MEMORY_Write	This routine provides interface to the file system to perform a media write operation in synchronous mode of the Memory driver.

Macros

	Name	Description
	DRV_MEMORY_COMMAND_HANDLE_INVALID	This value defines the Memory Driver's Invalid Command Handle.

Types

	Name	Description
	DRV_MEMORY_COMMAND_HANDLE	Handle to identify commands queued in the driver.
	DRV_MEMORY_TRANSFER_HANDLER	Pointer to a Memory Driver Event handler function

Description

Memory Driver Interface Definition

The Memory driver provides a simple interface to manage the MEMORYVF series of SQI Flash Memory connected to Microchip micro controllers. This file defines the interface definition for the Memory driver.

File Name

drv_memory.h

Company

Microchip Technology Inc.

drv_memory_definitions.h

Memory Driver Interface Definition

Enumerations

	Name	Description
	MEMORY_DEVICE_TRANSFER_STATUS	Memory Device Transfer Status.

Structures

	Name	Description
	DRV_MEMORY_DEVICE_INTERFACE	Memory Device API.

	DRV_MEMORY_INIT	Memory Driver Initialization Data
	MEMORY_DEVICE_GEOMETRY	Memory Device Geometry Table.

Types

	Name	Description
	DRV_MEMORY_DEVICE_CLOSE	Function pointer typedef to close the attached media
	DRV_MEMORY_DEVICE_EVENT_HANDLER_SET	Function pointer typedef to set the event handler with attached media
	DRV_MEMORY_DEVICE_GEOMETRY_GET	Function pointer typedef to get the Geometry details from attached media
	DRV_MEMORY_DEVICE_OPEN	Function pointer typedef to open the attached media
	DRV_MEMORY_DEVICE_PAGE_WRITE	Function pointer typedef to write a page to the attached media
	DRV_MEMORY_DEVICE_READ	Function pointer typedef to read from the attached media
	DRV_MEMORY_DEVICE_SECTOR_ERASE	Function pointer typedef to erase a sector from attached media
	DRV_MEMORY_DEVICE_STATUS	Function pointer typedef to get the status of the attached media
	DRV_MEMORY_DEVICE_TRANSFER_STATUS_GET	Function pointer typedef to get the transfer Status from attached media
	DRV_MEMORY_EVENT_HANDLER	Function pointer typedef for event handler to be sent to attached media

Description

Memory Driver Interface Definition

The Memory driver provides a simple interface to manage the various memory devices. This file defines the interface definition for the Memory driver.

File Name

drv_memory_definitions.h

Company

Microchip Technology Inc.

SD Card Host Controller Library

This section describes the SD Card Host Controller Driver Library.

Introduction

This Library provides an interface to use SD Card Host Controller Driver to communicate with SD-Card using HSMCI Peripheral.

Description

SDHC Driver is a multi-client single-instance buffer model based block driver interface which can be used to communicate with SD-Cards

Key Features of SDHC Driver:

1. It works only in Asynchronous modes of operation.
2. It works in both Bare Metal and RTOS environment.

Using the Library

This topic describes the basic architecture of the SDHC Driver and provides information on how it works.

Description

SDHC Driver can be used to communicate with SD-Cards.

Modes supported:

1. Asynchronous mode - Bare-metal and RTOS

It can be used in multiple ways

1. Application can directly use the SDHC driver API's to perform Read/Write operations.
2. Application can use File System service layer and perform file operations.
3. It can be interfaced with Middleware's like USB.

Abstraction Model

Provides abstraction of the SDHC Driver Library with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

SDHC Driver provides abstraction to communicate with SD-Card via HSMCI Interface..

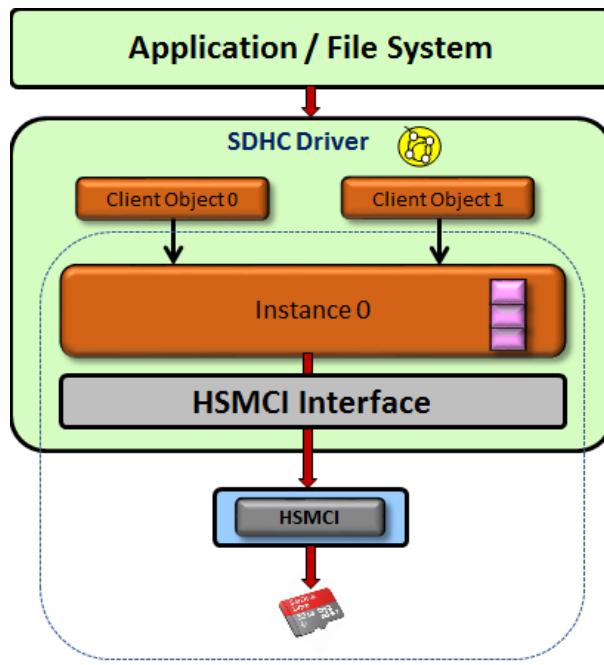


Figure-1 SDHC Driver Block Diagram

How the Library Works

This topic describes the basic architecture of the SDHC Driver Library.

Description

SDHC Library is a multi-client, single-instance buffer queue model based block driver interface.

SDHC Driver Features

- Driver has a buffer queue which allows the capability of accepting multiple requests.
- Driver can either have File-system as client or Application as client.
- Every transfer request expects data in blocks. Block details (**Size and number of blocks**) can be retrieved by

DRV_SDHC_GeometryGet()

- Driver provides feature to register call back for transfer complete event, which can be used by clients to get notified.
- When SDHC driver is connected to File System buffer queue is disabled as File system is blocking interface.
- Works in both Bare-Metal and RTOS environment in Asynchronous mode.
 - **Bare-Metal:**
 - A dedicated task routine **DRV_SDHC_Tasks()** is called from **SYS_Tasks()** to process the data from the instance queue
 - **RTOS:**
 - A dedicated thread is created for task routine **DRV_SDHC_Tasks()** to process the data from the instance queue
- API's return with a valid handle which can be used to check whether transfer request is accepted or not.
- A Client specific handler will be called to indicate the status of transfer.

Configuring the Library

This Section provides information on how to configure SDHC Driver library.

Description

SDHC Driver library should be configured via MHC. Below are the Snapshot of the MHC configuration window for SDHC driver and brief description.

User Configurations

1. Number Of SDHC Instances

1. Specifies number of SDHC Driver instances used. Currently maximum 1 instance is supported

2. Number Of SDHC Driver Clients

1. Specifies number of clients to access the specific instance of the driver.

3. Number of SDHC Buffer Objects

1. Specifies maximum number of requests to be buffered in queue.
2. When Connected to File system this value is set to 1 and non modifiable

4. Data Transfer Bus Width

1. Specifies the Bus width to be used for data transfer (1-Bit, 4-Bit)

5. Maximum Bus Speed

1. Specifies Bus Speed to be used for Communication with SD-card(DEFAULT_SPEED, HIGH_SPEED)
2. Standard SD-Cards only support Default Speed
3. SD-Cards with High Capacity support Both Default and High Speed

6. Use Write Protect (SDWP#) Pin

1. When Selected Write protect check is enabled.
2. A dedicated pin has to be configured from Pin settings

7. Use Card Detect (SDCD#) Pin

1. When Selected Dedicated pin can be used to detect the Card.
2. A dedicated pin has to be configured from Pin settings

8. DMA Channel for Transmit and Receive

1. Specifies the DMA Channel configured for Transmit and Receive

9. RTOS Settings

1. Stack Size
 - 1. Maximum Stack size to be allocated for SDHC Driver.
2. Task Priority
 - 1. Specifies Priority for the SDHC driver task thread
3. Task Delay
 - 1. Specifies Delay to sleep for the SDHC task Thread after every call.

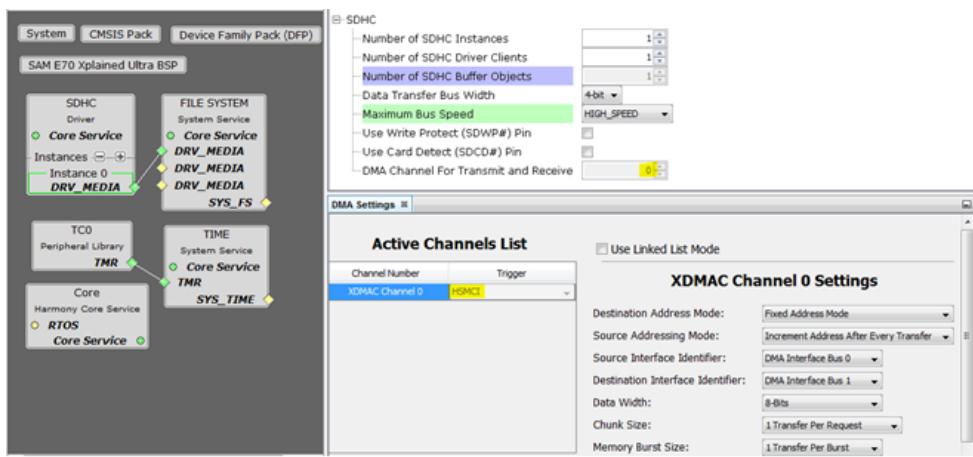


Figure-2 SDHC Driver with File System

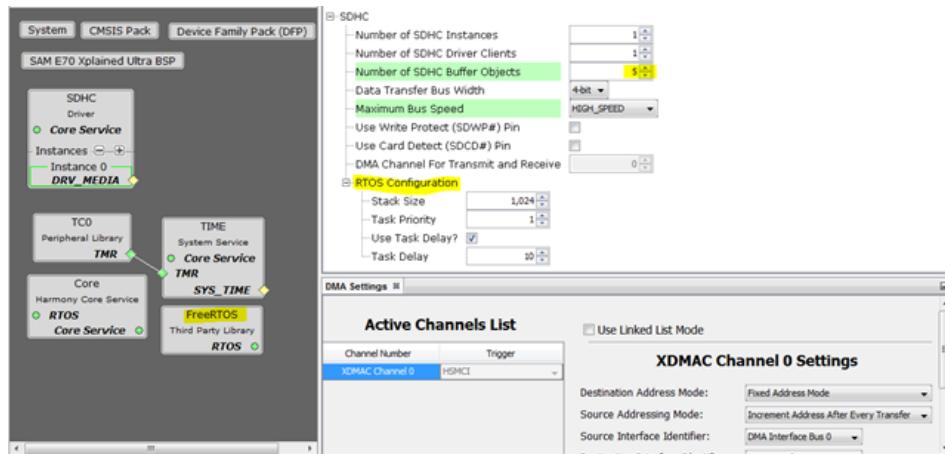


Figure-3 SDHC Driver without File System and With RTOS

Building the Library

This Section provides information on how the SDHC Driver Library can be built.

Description

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Functions

	Name	Description
≡	DRV_SDHC_Initialize	Initializes the SD Card driver.
≡	DRV_SDHC_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings.
≡	DRV_SDHC_Deinitialize	Deinitializes the specified instance of the SD Card driver module.
≡	DRV_SDHC_Tasks	Maintains the driver's state machine.

b) Core Client Functions

	Name	Description
≡	DRV_SDHC_Open	Opens the specified SD Card driver instance and returns a handle to it.
≡	DRV_SDHC_Close	Closes an opened-instance of the SD Card driver.
≡	DRV_SDHC_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

c) Block Operation Functions

	Name	Description
≡	DRV_SDHC_Status	Provides the current status of the SD Card driver module.
≡	DRV_SDHC_AsyncRead	Reads blocks of data from the specified block address of the SD Card.
≡	DRV_SDHC_AsyncWrite	Writes blocks of data starting at the specified address of the SD Card.
≡	DRV_SDHC_CommandStatusGet	Gets the current status of the command.

d) Media Interface Functions

	Name	Description
≡	DRV_SDHC_GeometryGet	Returns the geometry of the device.
≡	DRV_SDHC_IsAttached	Returns the physical attach status of the SD Card.
≡	DRV_SDHC_IsWriteProtected	Returns the write protect status of the SDHC.

e) Data Types and Constants

	Name	Description
	DRV_SDHC_BUS_WIDTH	SDHC Bus Width
	DRV_SDHC_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_SDHC_COMMAND_STATUS	SDHC Driver Command Events
	DRV_SDHC_EVENT	Identifies the possible events that can result from a request.
	DRV_SDHC_EVENT_HANDLER	SDHC Driver Event Handler Function Pointer
	DRV_SDHC_INIT	SD Host Controller Driver Initialization Data
	DRV_SDHC_SPEED_MODE	SDHC Bus Speed
	SDHC_DETECTION_LOGIC	SD Card Detection Logic
	DRV_SDHC_COMMAND_HANDLE_INVALID	SDHC Driver's Invalid Command Handle.
	DRV_SDHC_INDEX_0	SDHC Driver Module Index Numbers
	DRV_SDHC_INDEX_COUNT	SDHC Driver Module Index Count
	SDHC_MAX_LIMIT	SDHC Driver Maximum allowed limit

Description

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

Refer to each section for a detailed description.

a) System Functions**DRV_SDHC_Initialize Function**

Initializes the SD Card driver.

File

[drv_sdhc.h](#)

C

```
SYS_MODULE_OBJ DRV_SDHC_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 SDHC routine is called.

This routine should only be called once during system initialization unless [DRV_SDHC_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_SDHC_Status](#) operation. The system must use [DRV_SDHC_Status](#) to find out when the driver is in the ready state.

Preconditions

None.

Example

```
DRV_SDHC_INIT      init;
SYS_MODULE_OBJ     objectHandle;

// Populate the SD Card initialization structure

objectHandle = DRV_SDHC_Initialize(DRV_SDHC_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_SDHC_Initialize
(
const     SYS_MODULE_INDEX index,
const     SYS_MODULE_INIT * const init
);
```

DRV_SDHC_Reinitialize Function

Reinitializes the driver and refreshes any associated hardware settings.

File

[drv_sdhc.h](#)

C

```
void DRV_SDHC_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_SDHC_Status](#) operation. The system must use [DRV_SDHC_Status](#) to find out when the driver is in the ready state.

Preconditions

Function [DRV_SDHC_Initialize](#) must have been called before calling this routine and a valid [SYS_MODULE_OBJ](#) must have been returned.

Example

```

DRV_SDHC_INIT      init;
SYS_MODULE_OBJ     objectHandle; // Returned from DRV_SDHC_Initialize

// Update the required fields of the SD Card initialization structure

DRV_SDHC_Reinitialize (objectHandle, (SYS_MODULE_INIT*)&init);

```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SDHC_Initialize routine
init	Pointer to the initialization data structure

Function

```

void DRV_SDHC_Reinitialize
(
    SYS_MODULE_OBJ     object,
    const   SYS_MODULE_INIT * const init
);

```

DRV_SDHC_Deinitialize Function

Deinitializes the specified instance of the SD Card driver module.

File

[drv_sdhc.h](#)

C

```
void DRV_SDHC_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_SDHC_Status](#) operation. The system has to use [DRV_SDHC_Status](#) to check if the de-initialization is complete.

Preconditions

Function [DRV_SDHC_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_SDHC_Initialize
SYS_STATUS              status;

DRV_SDHC_Deinitialize(objectHandle);

status = DRV_SDHC_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_SDHC_Initialize routine.

Function

```
void DRV_SDHC_Deinitialize
(
    SYS_MODULE_OBJ object
);
```

DRV_SDHC_Tasks Function

Maintains the driver's state machine.

File

[drv_sdhc.h](#)

C

```
void DRV_SDHC_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_SDHC_Initialize](#) routine must have been called for the specified SDHC driver instance.

Example

```
SYS_MODULE_OBJ object;      // Returned from DRV_SDHC_Initialize

while (true)
{
    DRV_SDHC_Tasks (object);

    // Do other tasks
}
```

Parameters

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

Function

```
void DRV_SDHC_Tasks
(
    SYS_MODULE_OBJ object
);
```

b) Core Client Functions

DRV_SDHC_Open Function

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

File

[drv_sdhc.h](#)

C

```
DRV_HANDLE DRV_SDHC_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_SDHC_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_SDHC_Initialize](#) must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_SDHC_Open (DRV_SDHC_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_SDHC_Open
(
const    SYS_MODULE_INDEX drvIndex,
const    DRV_IO_INTENT   intent
);
```

DRV_SDHC_Close Function

Closes an opened-instance of the SD Card driver.

File

[drv_sdhc.h](#)

C

```
void DRV_SDHC_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_SDHC_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_SDHC_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

[DRV_SDHC_Open](#) must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle; // Returned from DRV_SDHC_Open

DRV_SDHC_Close (handle);
```

Parameters

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

Function

```
void DRV_SDHC_Close
(
    DRV_HANDLE handle
);
```

DRV_SDHC_EventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

File

[drv_sdhc.h](#)

C

```
void DRV_SDHC_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_SDHC_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_SDHC_COMMAND_HANDLE commandHandle;

// drvSDHCHandle is the handle returned
// by the DRV_SDHC_Open function.

// Client registers an event handler with driver. This is done once.

DRV_SDHC_EventHandlerSet(drvSDHCHandle, APP_SDHCEventHandler, (uintptr_t)&myAppObj);

DRV_SDHC_AsyncRead(drvSDHCHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SDHC_COMMAND_HANDLE_INVALID == commandHandle)
{
```

```

    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_SDHCEventHandler(DRV_SDHC_EVENT event,
    DRV_SDHC_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_SDHC_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SDHC_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_SDHC_EventHandlerSet
(
    const DRV_HANDLE handle,
    const void * eventHandler,
    const uintptr_t context
);

```

c) Block Operation Functions

DRV_SDHC_Status Function

Provides the current status of the SD Card driver module.

File

[drv_sdhc.h](#)

C

```
SYS_STATUS DRV_SDHC_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_SDHC_Initialize](#) must have been called before calling this

Example

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

status = DRV_SDHC_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_SDHC_Initialize routine

Function

```
SYS_STATUS DRV_SDHC_Status
(
    SYS_MODULE_OBJ object
);
```

DRV_SDHC_AsyncRead Function

Reads blocks of data from the specified block address of the SD Card.

File

[drv_sdhc.h](#)

C

```
void DRV_SDHC_AsyncRead(DRV_HANDLE handle, DRV_SDHC_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_SDHC_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_SDHC_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_SDHC_EVENT_COMMAND_COMPLETE](#) event if the buffer was processed successfully or [DRV_SDHC_EVENT_COMMAND_ERROR](#) event if the buffer was not processed successfully.

Remarks

None.

Preconditions

[DRV_SDHC_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_SDHC_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySDHCHandle is the handle returned
// by the DRV_SDHC_Open function.

DRV_SDHC_AsyncRead(mySDHCHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SDHC_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_SDHC_AsyncRead
(
    const DRV_HANDLE handle,
    DRV_SDHC_COMMAND_HANDLE * commandHandle,
    void * targetBuffer,
    uint32_t blockStart,
    uint32_t nBlock
);
```

DRV_SDHC_AsyncWrite Function

Writes blocks of data starting at the specified address of the SD Card.

File

[drv_sdhc.h](#)

C

```
void DRV_SDHC_AsyncWrite(DRV_HANDLE handle, DRV_SDHC_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_SDHC_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_SDHC_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_SDHC_EVENT_COMMAND_COMPLETE](#) event if the buffer was processed successfully or [DRV_SDHC_EVENT_COMMAND_ERROR](#) event if the buffer was not processed successfully.

Remarks

None.

Preconditions

DRV_SDHC_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_SDHC_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySDHCHandle is the handle returned
// by the DRV_SDHC_Open function.

// Client registers an event handler with driver

DRV_SDHC_EventHandlerSet(mySDHCHandle, APP_SDHCEventHandler, (uintptr_t)&myAppObj);

DRV_SDHC_AsyncWrite(mySDHCHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SDHC_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_SDHCEventHandler(DRV_SDHC_EVENT event,
                           DRV_SDHC_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SDHC_EVENT_COMMAND_COMPLETE:
            // This means the data was transferred.
            break;
        case DRV_SDHC_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_SDHC_AsyncWrite

```

(
const    DRV_HANDLE handle,
DRV_SDHC_COMMAND_HANDLE * commandHandle,
void * sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
);

```

DRV_SDHC_CommandStatusGet Function

Gets the current status of the command.

File

[drv_sdhc.h](#)

C

```
DRV_SDHC_COMMAND_STATUS DRV_SDHC_CommandStatusGet(const DRV_HANDLE handle, const
DRV_SDHC_COMMAND_HANDLE commandHandle);
```

Returns

- A [DRV_SDHC_COMMAND_STATUS](#) value describing the current status of the command.
- [DRV_SDHC_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_SDHC_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_SDHC_Open\(\)](#) must have been called to obtain a valid opened device handle.

Example

```

DRV_HANDLE           handle;           // Returned from DRV_SDHC_Open
DRV_SDHC_COMMAND_HANDLE commandHandle;
DRV_SDHC_COMMAND_STATUS status;

status = DRV_SDHC_CommandStatusGet(handle, commandHandle);
if(status == DRV_SDHC_COMMAND_COMPLETED)
{
    // Operation Done
}

```

Parameters

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

Function

```

DRV_SDHC_COMMAND_STATUS DRV_SDHC_CommandStatusGet
(
const    DRV_HANDLE handle,
const    DRV_SDHC_COMMAND_HANDLE commandHandle

```

);

d) Media Interface Functions**DRV_SDHC_GeometryGet Function**

Returns the geometry of the device.

File

[drv_sdhc.h](#)

C

```
SYS_MEDIA_GEOMETRY * DRV_SDHC_GeometryGet(const DRV_HANDLE handle);
```

Returns

[SYS_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_SDHC_Open\(\)](#) routine must have been called to obtain a valid opened device handle.

Example

```
SYS_MEDIA_GEOMETRY * SDHCGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalSize;

SDHCGeometry = DRV_SDHC_GeometryGet(SDHCOpenHandle1);

readBlockSize = SDHCGeometry->geometryTable->blockSize;
nReadBlocks = SDHCGeometry->geometryTable->numBlocks;
nReadRegions = SDHCGeometry->numReadRegions;

writeBlockSize = (SDHCGeometry->geometryTable +1)->blockSize;
eraseBlockSize = (SDHCGeometry->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\_MEDIA\_GEOMETRY * DRV_SDHC_GeometryGet
(
  const DRV\_HANDLE handle
);
```

DRV_SDHC_IsAttached Function

Returns the physical attach status of the SD Card.

File

[drv_sdhc.h](#)

C

```
bool DRV_SDHC_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 SDHC driver otherwise returns false.

Description

This function returns the physical attach status of the SD Card.

Remarks

None.

Preconditions

The [DRV_SDHC_Open\(\)](#) routine must have been called to obtain a valid opened device handle.

Example

```
bool isSDHCAttached;
isSDHCAttached = DRV_SDHC_isAttached(drvSDHCHandle);
```

Parameters

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

Function

```
bool DRV_SDHC_IsAttached
(
const DRV_HANDLE handle
);
```

DRV_SDHC_IsWriteProtected Function

Returns the write protect status of the SDHC.

File

[drv_sdhc.h](#)

C

```
bool DRV_SDHC_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 SDHC. This function returns true if the SD Card is write protected otherwise

it returns false.

Remarks

None.

Preconditions

The [DRV_SDHC_Open\(\)](#) routine must have been called to obtain a valid opened device handle.

Example

```
bool isWriteProtected;
isWriteProtected = DRV_SDHC_IsWriteProtected(drvSDHCHandle);
```

Parameters

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

Function

```
bool DRV_SDHC_IsWriteProtected
(
const DRV_HANDLE handle
);
```

e) Data Types and Constants

DRV_SDHC_BUS_WIDTH Enumeration

SDHC Bus Width

File

[drv_sdhc.h](#)

C

```
typedef enum {
DRV_SDHC_BUS_WIDTH_1_BIT = 0,
DRV_SDHC_BUS_WIDTH_4_BIT
} DRV_SDHC_BUS_WIDTH;
```

Description

This enum defines SDHC Transfer bus widths.

Remarks

None.

DRV_SDHC_COMMAND_HANDLE Type

Handle identifying commands queued in the driver.

File

[drv_sdhc.h](#)

C

```
typedef SYS_MEDIA_BLOCK_COMMAND_HANDLE DRV_SDHC_COMMAND_HANDLE;
```

Description

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_SDHC_COMMAND_STATUS Enumeration

SDHC Driver Command Events

File

[drv_sdhc.h](#)

C

```
typedef enum {
    DRV_SDHC_COMMAND_COMPLETED = SYS_MEDIA_COMMAND_COMPLETED,
    DRV_SDHC_COMMAND_QUEUED = SYS_MEDIA_COMMAND_QUEUED,
    DRV_SDHC_COMMAND_IN_PROGRESS = SYS_MEDIA_COMMAND_IN_PROGRESS,
    DRV_SDHC_COMMAND_ERROR_UNKNOWN = SYS_MEDIA_COMMAND_UNKNOWN
} DRV_SDHC_COMMAND_STATUS;
```

Members

Members	Description
DRV_SDHC_COMMAND_COMPLETED = SYS_MEDIA_COMMAND_COMPLETED	Done OK and ready
DRV_SDHC_COMMAND_QUEUED = SYS_MEDIA_COMMAND_QUEUED	Scheduled but not started
DRV_SDHC_COMMAND_IN_PROGRESS = SYS_MEDIA_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_SDHC_COMMAND_ERROR_UNKNOWN = SYS_MEDIA_COMMAND_UNKNOWN	Unknown Command

Description

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_SDHC_EventHandlerSet](#) function when a request is completed.

DRV_SDHC_EVENT Enumeration

Identifies the possible events that can result from a request.

File

[drv_sdhc.h](#)

C

```
typedef enum {
    DRV_SDHC_EVENT_COMMAND_COMPLETE = SYS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE,
    DRV_SDHC_EVENT_COMMAND_ERROR = SYS_MEDIA_EVENT_BLOCK_COMMAND_ERROR
```

```
    } DRV_SDHC_EVENT;
```

Members

Members	Description
DRV_SDHC_EVENT_COMMAND_COMPLETE = SYS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE	Operation has been completed successfully.
DRV_SDHC_EVENT_COMMAND_ERROR = SYS_MEDIA_EVENT_BLOCK_COMMAND_ERROR	There was an error during the operation

Description

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_SDHC_EventHandlerSet](#) function when a request is completed.

DRV_SDHC_EVENT_HANDLER Type

SDHC Driver Event Handler Function Pointer

File

[drv_sdhc.h](#)

C

```
typedef SYS_MEDIA_EVENT_HANDLER DRV_SDHC_EVENT_HANDLER;
```

Returns

None.

Description

This data type defines the required function signature for the SDHC 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_SDHC_EVENT_COMMAND_COMPLETE, it means that the write or a erase operation was completed successfully.

If the event is DRV_SDHC_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_SDHC_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_MySDHCEventHandler
(
    DRV_SDHC_EVENT event,
    DRV_SDHC_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SDHC_EVENT_COMMAND_COMPLETE:
```

```

    // Handle the completed buffer.
    break;

    case DRV_SDHC_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

DRV_SDHC_INIT Structure

SD Host Controller Driver Initialization Data

File

[drv_sdhc.h](#)

C

```

typedef struct {
    bool sdCardDetectEnable;
    bool sdWriteProtectEnable;
    DRV_SDHC_SPEED_MODE speedMode;
    DRV_SDHC_BUS_WIDTH busWidth;
    bool isFsEnabled;
} DRV_SDHC_INIT;

```

Description

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_SDHC_Initialize](#) routine.

DRV_SDHC_SPEED_MODE Enumeration

SDHC Bus Speed

File

[drv_sdhc.h](#)

C

```

typedef enum {
    DRV_SDHC_SPEED_MODE_DEFAULT = 0,
    DRV_SDHC_SPEED_MODE_HIGH
} DRV_SDHC_SPEED_MODE;

```

Description

This enum defines SDHC bus Speeds Supported.

- Standard SD-Cards only support Default Speed.
- SD-Cards with High Capacity support Both Default and High Speed

Remarks

None.

SDHC_DETECTION_LOGIC Enumeration

SD Card Detection Logic

File

[drv_sdhc.h](#)

C

```
typedef enum {
    SDHC_DETECTION_LOGIC_ACTIVE_LOW,
    SDHC_DETECTION_LOGIC_ACTIVE_HIGH
} SDHC_DETECTION_LOGIC;
```

Members

Members	Description
SDHC_DETECTION_LOGIC_ACTIVE_LOW	The media event is SD Card attach
SDHC_DETECTION_LOGIC_ACTIVE_HIGH	The media event is SD Card detach

Description

This enum defines Card detection logic levels.

Remarks

None.

DRV_SDHC_COMMAND_HANDLE_INVALID Macro

SDHC Driver's Invalid Command Handle.

File

[drv_sdhc.h](#)

C

```
#define DRV_SDHC_COMMAND_HANDLE_INVALID SYS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID
```

Description

This value defines the SDHC Driver Invalid Command Handle. This value is returned by read or write routines when the command request was not accepted.

Remarks

None.

DRV_SDHC_INDEX_0 Macro

SDHC Driver Module Index Numbers

File

[drv_sdhc.h](#)

C

```
#define DRV_SDHC_INDEX_0 (0)
```

Description

These constants provide SDHC driver index definitions.

Remarks

These values should be passed into the [DRV_SDHC_Initialize](#) and [DRV_SDHC_Open](#) routines to identify the driver instance in use.

DRV_SDHC_INDEX_COUNT Macro

SDHC Driver Module Index Count

File

```
drv_sdhc.h
```

C

```
#define DRV_SDHC_INDEX_COUNT (1)
```

Description

This constant identifies number of valid SDHC driver indices.

Remarks

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

SDHC_MAX_LIMIT Macro

SDHC Driver Maximum allowed limit

File

```
drv_sdhc.h
```

C

```
#define SDHC_MAX_LIMIT (2)
```

Description

This constant identifies number of valid SD Card driver indices.

Remarks

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

Files**Files**

Name	Description
drv_sdhc.h	SD Host Controller Driver Interface File

Description

This section will list only the library's interface header file(s).

drv_sdhc.h

SD Host Controller Driver Interface File

Enumerations

	Name	Description
	DRV_SDHC_BUS_WIDTH	SDHC Bus Width
	DRV_SDHC_COMMAND_STATUS	SDHC Driver Command Events
	DRV_SDHC_EVENT	Identifies the possible events that can result from a request.
	DRV_SDHC_SPEED_MODE	SDHC Bus Speed
	SDHC_DETECTION_LOGIC	SD Card Detection Logic

Functions

	Name	Description
≡	DRV_SDHC_AsyncRead	Reads blocks of data from the specified block address of the SD Card.
≡	DRV_SDHC_AsyncWrite	Writes blocks of data starting at the specified address of the SD Card.
≡	DRV_SDHC_Close	Closes an opened-instance of the SD Card driver.
≡	DRV_SDHC_CommandStatusGet	Gets the current status of the command.
≡	DRV_SDHC_Deinitialize	Deinitializes the specified instance of the SD Card driver module.
≡	DRV_SDHC_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.
≡	DRV_SDHC_GeometryGet	Returns the geometry of the device.
≡	DRV_SDHC_Initialize	Initializes the SD Card driver.
≡	DRV_SDHC_IsAttached	Returns the physical attach status of the SD Card.
≡	DRV_SDHC_IsWriteProtected	Returns the write protect status of the SDHC.
≡	DRV_SDHC_Open	Opens the specified SD Card driver instance and returns a handle to it.
≡	DRV_SDHC_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings.
≡	DRV_SDHC_Status	Provides the current status of the SD Card driver module.
≡	DRV_SDHC_Tasks	Maintains the driver's state machine.

Macros

	Name	Description
	DRV_SDHC_COMMAND_HANDLE_INVALID	SDHC Driver's Invalid Command Handle.
	DRV_SDHC_INDEX_0	SDHC Driver Module Index Numbers
	DRV_SDHC_INDEX_COUNT	SDHC Driver Module Index Count
	SDHC_MAX_LIMIT	SDHC Driver Maximum allowed limit

Structures

	Name	Description
	DRV_SDHC_INIT	SD Host Controller Driver Initialization Data

Types

	Name	Description
	DRV_SDHC_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_SDHC_EVENT_HANDLER	SDHC Driver Event Handler Function Pointer

Description

SD Host Controller Driver Interface

The SD Host Controller driver provides a simple interface to manage the SD Host Controller peripheral. This file defines the interface definitions and prototypes for the SD Host Controller driver.

File Name

drv_sdhc.h

Company

Microchip Technology Inc.

SPI Driver Library Help

This section describes the SPI Driver Library.

Introduction

This library provides an interface to manage the Serial Peripheral Interface (SPI) module.

Description

SPI Driver is a multi-client, multi-instance buffer model based driver interface which can be used to communicate with various slave devices like EEPROM, ADC etc.

Key Features of SPI Driver:

1. **Multi Client:** Each instance of SPI driver can have multiple clients. Which effectively means there can be more than one slave device connected on the same SPI bus and SPI driver can handle it effectively.
2. **Multi Instance:** Provides interface to manage multiple instances of SPI peripheral effectively.
3. Supports Asynchronous and Synchronous modes of operation.
4. Supports Bare Metal and RTOS environments.
5. Supports DMA.

Using the Library

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

Description

SPI Driver can be used in 2 different modes:

1. Asynchronous Mode
2. Synchronous Mode

Following table describes which function is supported in which mode of SPI Driver:

Functions	Asynchronous Mode	Synchronous Mode
DRV_SPI_Initialize	Yes	Yes
DRV_SPI_Status	Yes	Yes
DRV_SPI_Open	Yes	Yes
DRV_SPI_Close	Yes	Yes
DRV_SPI_TransferSetup	Yes	Yes
DRV_SPI_WriteReadTransferAdd	Yes	No
DRV_SPI_WriteTransferAdd	Yes	No
DRV_SPI_ReadTransferAdd	Yes	No
DRV_SPI_TransferEventHandlerSet	Yes	No
DRV_SPI_TransferStatusGet	Yes	No
DRV_SPI_WriteTransfer	No	Yes

DRV_SPI_ReadTransfer	No	Yes
DRV_SPI_WriteReadTransfer	No	Yes

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.

Description

The SPI driver provides interface to use multiple SPI Peripherals (via SPI PLIB instances) connected with one or multiple slave devices.

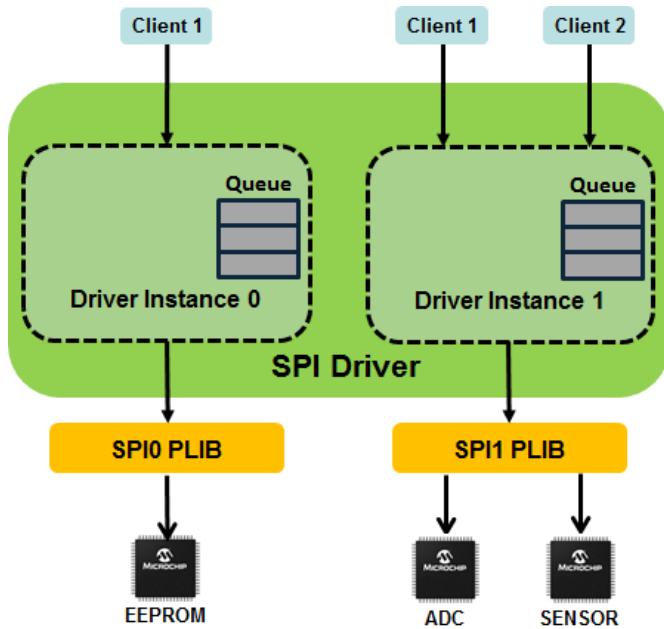


Figure-1 SPI Driver Abstraction Model

Note: Queue is not present in Synchronous mode of the driver.

Asynchronous Mode

This chapter describes how to use SPI Driver in Asynchronous mode.

Description

Asynchronous mode of SPI driver allows user to use the library in non-blocking manner. This mode is supported in RTOS and baremetal both the environments. Transfer APIs of this mode return a valid handle. Application can check the data transfer completion status in two ways:

1. **Callback:** Callback can be registered with the driver using [DRV_SPI_TransferEventHandlerSet](#) API and it needs to be registered prior to calling the transfer API . Here is the example of SPI write using callback:

```
DRV_SPI_TRANSFER_HANDLE transferHandle1;
DRV_SPI_TRANSFER_SETUP setup;
DRV_HANDLE           drvSPIHandle;
uint8_t              writeData[] = "SPI Driver Async Mode Callback";

void SpiEventHandler (DRV_SPI_TRANSFER_EVENT event, DRV_SPI_TRANSFER_HANDLE transferHandle,
uintptr_t context )
{
```

```

    if (event == DRV_SPI_TRANSFER_EVENT_COMPLETE)
    {
        if(transferHandle == transferHandle1)
        {
            // transfer1 is completed
        }
    }
    else
    {
        // transfer1 had error.
    }
}

// SPI Driver Initialization is done in MHC generated code

/* Setup structure for SPI transfer */
setup.baudRateInHz = 600000;
setup.clockPhase = DRV_SPI_CLOCK_PHASE_VALID.LEADING_EDGE;
setup.clockPolarity = DRV_SPI_CLOCK_POLARITY_IDLE_LOW;
setup.dataBits = DRV_SPI_DATA_BITS_8;
setup.chipSelect = SYS_PORT_PIN_PD16;
setup.csPolarity = DRV_SPI_CS_POLARITY_ACTIVE_LOW;

/* Open the SPI Driver*/
drvSPIHandle = DRV_SPI_Open( DRV_SPI_INDEX_0, DRV_IO_INTENT_READWRITE);

if(drvSPIHandle != DRV_HANDLE_INVALID)
{
    if(DRV_SPI_TransferSetup(drvSPIHandle, &setup) == true)
    {
        // setup is successful
    }
    DRV_SPI_TransferEventHandlerSet(drvSPIHandle, SpiEventHandler, (uintptr_t)NULL);
}

DRV_SPI_WriteTransferAdd(drvSPIHandle, &writeData, sizeof(writeData), &transferHandle1 );
if(transferHandle1 == DRV_SPI_TRANSFER_HANDLE_INVALID)
{
    
}
else
{
    
"SpiEventHandler" will be called when transfer is completed. */
}

```

2. **Polling:** Data transfer status polling can be done using [DRV_SPI_TransferStatusGet](#) API. Here is the example of SPI write using status polling:

```

DRV_SPI_TRANSFER_HANDLE transferHandle1;
DRV_SPI_TRANSFER_SETUP setup;
DRV_HANDLE           drvSPIHandle;
uint8_t              writeData[] = "SPI Driver Async Mode Polling";

// SPI Driver Initialization is done in MHC generated code

/* Setup structure for SPI transfer */
setup.baudRateInHz = 600000;
setup.clockPhase = DRV_SPI_CLOCK_PHASE_VALID.LEADING_EDGE;
setup.clockPolarity = DRV_SPI_CLOCK_POLARITY_IDLE_LOW;
setup.dataBits = DRV_SPI_DATA_BITS_8;
setup.chipSelect = SYS_PORT_PIN_PD16;
setup.csPolarity = DRV_SPI_CS_POLARITY_ACTIVE_LOW;

/* Open the SPI Driver*/
drvSPIHandle = DRV_SPI_Open( DRV_SPI_INDEX_0, DRV_IO_INTENT_READWRITE);

```

```

if(drvSPIHandle != DRV_HANDLE_INVALID)
{
    if(DRV_SPI_TransferSetup(drvSPIHandle, &setup) == true)
    {
        // setup is successful
    }
}

DRV_SPI_WriteTransferAdd(drvSPIHandle, &writeData, sizeof(writeData), &transferHandle1 );
if(transferHandle1 == DRV_SPI_TRANSFER_HANDLE_INVALID)
{
    /* transfer request was not successful, try again */
}
else
{
    /* transfer request was successful, transfer status can be
polled anytime using DRV_SPI_TransferStatusGet API */
}

if (DRV_SPI_TransferStatusGet(transferHandle1) == DRV_SPI_TRANSFER_EVENT_COMPLETE)
{
    // transfer1 is completed successfully
}

```

Synchronous mode

This chapter describes how to use SPI Driver in Synchronous mode.

Description

Synchronous mode of SPI driver allows user to use the library in blocking manner. This mode is supported in RTOS environment only. Transfer APIs of this mode return true or false depending on transfer status. There is no need to check the transfer status separately. Here is the example of SPI write in synchronous mode:

```

DRV_SPI_TRANSFER_SETUP    setup;
DRV_HANDLE                drvSPIHandle;
uint8_t                   writeData[] = "SPI Driver Async Mode";

// SPI Driver Initialization is done in MHC generated code

/* Setup structure for SPI transfer */
setup.baudRateInHz = 600000;
setup.clockPhase = DRV_SPI_CLOCK_PHASE_VALID.LEADING_EDGE;
setup.clockPolarity = DRV_SPI_CLOCK_POLARITY_IDLE_LOW;
setup.dataBits = DRV_SPI_DATA_BITS_8;
setup.chipSelect = SYS_PORT_PIN_PD16;
setup.csPolarity = DRV_SPI_CS_POLARITY_ACTIVE_LOW;

/* Open the SPI Driver*/
drvSPIHandle = DRV_SPI_Open( DRV_SPI_INDEX_0, DRV_IO_INTENT_READWRITE);

if(drvSPIHandle != DRV_HANDLE_INVALID)
{
    if(DRV_SPI_TransferSetup(drvSPIHandle, &setup) == true)
    {
        // setup is successful
    }
}

if(DRV_SPI_WriteTransfer(drvSPIHandle, &writeData, sizeof(writeData)) == true)
{
    // transfer is completed successfully
}

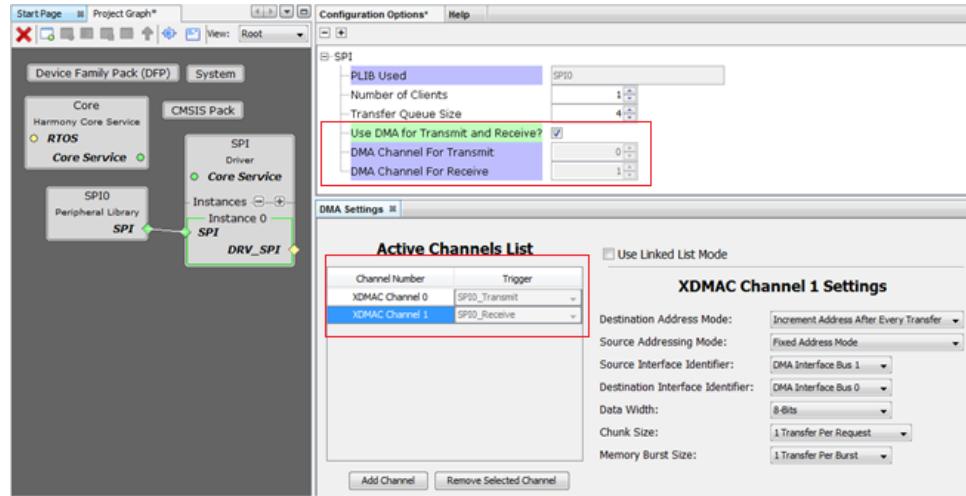
```

Configuring the Library

This Section provides information on how to configure SPI Driver library.

Description

SPI Driver library should be configured via MHC. Below is the Snapshot of the MHC configuration window for SPI driver and brief description.



Common User Configuration for all the Driver Instances:

- **Driver Mode:**
 1. Allows User to select the mode of driver (**Asynchronous or Synchronous**). This setting is common for all the instances

Driver Instance Specific User Configurations

- **Number Of Clients:**
 - Specify number of clients for the specific instance of the driver.
- **Transfer Queue Size:**
 - In Asynchronous mode, specify maximum number of transfer requests which can be queued for the specific instance of the driver.
 - **In Synchronous mode, buffer Queuing is disabled.**
- **Use DMA for Transmit and Receive:**
 - This option is used if DMA mode is intended for specific instance.
 - **DMA Channel For Transmit**
 - DMA Channel used for transmission is auto allocated based on availability in DMA Configuration
 - **DMA Channel For Receive**
 - DMA Channel used for Receiving is auto allocated based on availability in DMA Configuration

Building the Library

This Section provides information on how the SPI Driver Library can be built.

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Functions

	Name	Description
≡	DRV_SPI_Initialize	Initializes the SPI instance for the specified driver index.
≡	DRV_SPI_Status	Gets the current status of the SPI driver module.

b) Client Setup Functions

	Name	Description
≡	DRV_SPI_Open	Opens the specified SPI driver instance and returns a handle to it.
≡	DRV_SPI_Close	Closes an opened instance of the SPI driver.
≡	DRV_SPI_TransferSetup	Sets the dynamic configuration of the driver including chip select pin.
≡	DRV_SPI_TransferEventHandlerSet	Allows a client to set a transfer event handling function for the driver to call back when queued transfer has finished.

c) Transfer Functions

	Name	Description
≡	DRV_SPI_ReadTransfer	This is a blocking function that receives data over SPI.
≡	DRV_SPI_WriteTransfer	This is a blocking function that transmits data over SPI.
≡	DRV_SPI_WriteReadTransfer	This is a blocking function that transmits and receives data over SPI.
≡	DRV_SPI_ReadTransferAdd	Queues a read operation.
≡	DRV_SPI_WriteTransferAdd	Queues a write operation.
≡	DRV_SPI_WriteReadTransferAdd	Queues a write-read transfer operation.

d) Transfer Status Functions

	Name	Description
≡	DRV_SPI_TransferStatusGet	Returns transfer add request status.

e) Data Types and Constants

	Name	Description
	DRV_SPI_TRANSFER_EVENT	Identifies the possible events that can result from a transfer add request.
	DRV_SPI_TRANSFER_HANDLE	Handle identifying the transfer request queued.
	DRV_SPI_TRANSFER_EVENT_HANDLER	Pointer to a SPI Driver Transfer Event handler function
	DRV_SPI_TRANSFER_HANDLE_INVALID	Definition of an invalid transfer 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.
	SYS_MODULE_INDEX	Identifies which instance of a system module should be initialized or opened.
	SYS_MODULE_OBJ	Handle to an instance of a system module.
	SYS_STATUS	Identifies the current status/state of a system module (including device drivers).
	SYS_MODULE_OBJ_INVALID	Object handle value returned if unable to initialize the requested instance of a system module.
	MAIN_RETURN	Defines the correct return type for the "main" routine.
	MAIN_RETURN_CODE	Casts the given value to the correct type for the return code from "main".
	SYS_ASSERT	Implements default system assert routine, asserts that "test" is true.
	SYS_MODULE_DEINITIALIZE_ROUTINE	Pointer to a routine that deinitializes a system module (driver, library, or system-maintained application).

	SYS_MODULE_INITIALIZE_ROUTINE	Pointer to a routine that initializes a system module (driver, library, or system-maintained application).
	SYS_MODULE_REINITIALIZE_ROUTINE	Pointer to a routine that reinitializes a system module (driver, library, or system-maintained application)
	SYS_MODULE_STATUS_ROUTINE	Pointer to a routine that gets the current status of a system module (driver, library, or system-maintained application).
	SYS_MODULE_TASKS_ROUTINE	Pointer to a routine that performs the tasks necessary to maintain a state machine in a module system module (driver, library, or system-maintained application).
	SYS_MODULE_OBJ_STATIC	Object handle value returned by static modules.

Description

This section describes the API functions of the SPI Driver library.

Refer to each section for a detailed description.

a) System Functions

DRV_SPI_Initialize Function

Initializes the SPI instance for the specified driver index.

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. Otherwise, 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.

Remarks

- This routine must be called before any other SPI routine is called.
- This routine must only be called once during system initialization.
- This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
// The following code snippet shows an example SPI driver initialization.
SYS_MODULE_OBJ objectHandle;

const DRV_SPI_PLIB_INTERFACE drvSPI0PlibAPI = {

    // SPI PLIB Setup
    .setup = (DRV_SPI_PLIB_SETUP)SPI0_TransferSetup,

    // SPI PLIB WriteRead function
}
```

```

.writeRead = (DRV_SPI_PLIB_WRITE_READ)SPI0_WriteRead,
// SPI PLIB Transfer Status function
.isBusy = (DRV_SPI_PLIB_IS_BUSY)SPI0_IsBusy,
// SPI PLIB Callback Register
.callbackRegister = (DRV_SPI_PLIB_CALLBACK_REGISTER)SPI0_CallbackRegister,
};

const DRV_SPI_INIT drvSPI0InitData = {
// SPI PLIB API
.spiplib = &drvSPI0PlibAPI,
.remapDataBits = drvSPI0remapDataBits,
.remapClockPolarity = drvSPI0remapClockPolarity,
.remapClockPhase = drvSPI0remapClockPhase,
/// SPI Number of clients
.numClients = DRV_SPI_CLIENTS_NUMBER_IDX0,
// SPI Client Objects Pool
.clientObjPool = (uintptr_t)&drvSPI0ClientObjPool[0],
// DMA Channel for Transmit
.dmaChannelTransmit = DRV_SPI_XMIT_DMA_CH_IDX0,
// DMA Channel for Receive
.dmaChannelReceive = DRV_SPI_RCV_DMA_CH_IDX0,
// SPI Transmit Register
.spiTransmitAddress = (void *)&(SPI0_REGS->SPI_TDR),
// SPI Receive Register
.spiReceiveAddress = (void *)&(SPI0_REGS->SPI_RDR),
// Interrupt source is DMA
.interruptSource = XDMAC IRQn,
// SPI Queue Size
.queueSize = DRV_SPI_QUEUE_SIZE_IDX0,
// SPI Transfer Objects Pool
.transferObjPool = (uintptr_t)&drvSPI0TransferObjPool[0],
};

objectHandle = DRV_SPI_Initialize(DRV_SPI_INDEX_0, (SYS_MODULE_INIT*)&drvSPI0InitData);
if (objectHandle == SYS_MODULE_OBJ_INVALID)
{
    // Handle error
}

```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to the init data structure containing any data necessary to initialize the driver.

Function

```

SYS_MODULE_OBJ DRV_SPI_Initialize
(

```

```
const    SYS_MODULE_INDEX index,
const    SYS_MODULE_INIT * const init
)
```

DRV_SPI_Status Function

Gets the current status of the SPI driver module.

File

[drv_spi.h](#)

C

```
SYS_STATUS DRV_SPI_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Initialization have succeeded and the SPI is ready for additional operations
- SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

Description

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

Remarks

A driver can be opened only when its status is SYS_STATUS_READY.

Preconditions

Function [DRV_SPI_Initialize](#) should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;      // Returned from DRV_SPI_Initialize
SYS_STATUS         spiStatus;

spiStatus = DRV_SPI_Status(object);
if (spiStatus == SYS_STATUS_READY)
{
    // This means now the driver can be opened using the
    // DRV_SPI_Open() function.
}
```

Parameters

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

Function

[SYS_STATUS](#) [DRV_SPI_Status](#)([SYS_MODULE_OBJ](#) object)

b) Client Setup Functions

DRV_SPI_Open Function

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

File

[drv_spi.h](#)

C

```
DRV_HANDLE DRV_SPI_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_SPI_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 peripheral 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 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.

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.

Preconditions

Function [DRV_SPI_Initialize](#) 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 (handle == DRV_HANDLE_INVALID)
{
    // 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_SPI_Open
(
    const SYS_MODULE_INDEX index,
    const DRV_IO_INTENT ioIntent
)
```

DRV_SPI_Close Function

Closes an opened-instance of the SPI driver.

File

[drv_spi.h](#)

C

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

Returns

None.

Description

This routine closes an opened-instance of the SPI driver, invalidating the handle. User should make sure that there is no transfer request pending before calling this API. A new handle must be obtained by calling [DRV_SPI_Open](#) before the caller may use the driver again.

Remarks

None.

Preconditions

[DRV_SPI_Open](#) must have been called to obtain a valid opened device handle.

Example

```
// 'handle', returned from the 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_TransferSetup Function

Sets the dynamic configuration of the driver including chip select pin.

File

[drv_spi.h](#)

C

```
bool DRV_SPI_TransferSetup(DRV_HANDLE handle, DRV_SPI_TRANSFER_SETUP * setup);
```

Returns

None.

Description

This function is used to update any of the DRV_SPI_TRANSFER_SETUP parameters for the selected client of the driver dynamically. For single client scenario, if GPIO has to be used for chip select, then calling this API with appropriate GPIO pin information becomes mandatory. For multi client scenario where different clients need different setup like baud rate, clock settings, chip select etc, then also calling this API is mandatory.

Note that all the elements of setup structure must be filled appropriately before using this API.

Remarks

None.

Preconditions

[DRV_SPI_Open](#) must have been called to obtain a valid opened device handle.

Example

```
// mySPIHandle is the handle returned by the DRV_SPI_Open function.
DRV_SPI_TRANSFER_SETUP setup;

setup.baudRateInHz = 10000000;
setup.clockPhase = DRV_SPI_CLOCK_PHASE_TRAILING_EDGE;
setup.clockPolarity = DRV_SPI_CLOCK_POLARITY_IDLE_LOW;
setup.dataBits = DRV_SPI_DATA_BITS_16;
setup.chipSelect = SYS_PORT_PIN_PC5;
setup.csPolarity = DRV_SPI_CS_POLARITY_ACTIVE_LOW;

DRV_SPI_TransferSetup ( mySPIHandle, &setup );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
*setup	A structure containing the new configuration settings

Function

bool DRV_SPI_TransferSetup ([DRV_HANDLE](#) handle, [DRV_SPI_TRANSFER_SETUP](#) * setup)

DRV_SPI_TransferEventHandlerSet Function

Allows a client to set a transfer event handling function for the driver to call back when queued transfer has finished.

File

[drv_spi.h](#)

C

```
void DRV_SPI_TransferEventHandlerSet(const DRV_HANDLE handle, const
DRV_SPI_TRANSFER_EVENT_HANDLER eventHandler, uintptr\_t context);
```

Returns

None.

Description

This function allows a client to register a transfer event handling function with the driver to call back when queued transfer has finished. When a client calls either the [DRV_SPI_ReadTransferAdd](#) or [DRV_SPI_WriteTransferAdd](#) or [DRV_SPI_WriteReadTransferAdd](#) function, it is provided with a handle identifying the transfer request that was added to the driver's queue. The driver will pass this handle back to the client by calling "eventHandler" function when the transfer has completed.

The event handler should be set before the client performs any "transfer 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 transfer request has completed, it does not need to register a callback.

Preconditions

[DRV_SPI_Open](#) must have been called to obtain a valid open instance handle.

Example

```
// myAppObj is an application specific state data object.
MY\_APP\_OBJ myAppObj;
```

```

uint8_t myTxBuffer[MY_TX_BUFFER_SIZE];
uint8_t myRxBuffer[MY_RX_BUFFER_SIZE];
DRV_SPI_TRANSFER_HANDLE transferHandle;

// mySPIHandle is the handle returned by the DRV_SPI_Open function.

// Client registers an event handler with driver. This is done once

DRV_SPI_TransferEventHandlerSet( mySPIHandle, APP_SPITransferEventHandler,
                                  (uintptr_t)&myAppObj );

DRV_SPI_WriteReadTransferAdd(mySPIHandle, myTxBuffer,
                             MY_TX_BUFFER_SIZE, myRxBuffer,
                             MY_RX_BUFFER_SIZE, &transferHandle);

if(transferHandle == DRV_SPI_TRANSFER_HANDLE_INVALID)
{
    // Error handling here
}

// Event is received when the transfer is completed.

void APP_SPITransferEventHandler(DRV_SPI_TRANSFER_EVENT event,
                                 DRV_SPI_TRANSFER_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_SPI_TRANSFER_EVENT_COMPLETE:
            // This means the data was transferred.
            break;

        case DRV_SPI_TRANSFER_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_SPI_TransferEventHandlerSet
(
    const DRV_HANDLE handle,
    const DRV_SPI_TRANSFER_EVENT_HANDLER eventHandler,
    const uintptr_t context
)

```

c) Transfer Functions

DRV_SPI_ReadTransfer Function

This is a blocking function that receives data over SPI.

File

[drv_spi.h](#)

C

```
bool DRV_SPI_ReadTransfer(const DRV_HANDLE handle, void* pReceiveData, size_t rxSize);
```

Returns

- true - receive is successful
- false - error has occurred

Description

This function does a blocking read operation. The function blocks till the data receive is complete. Function will return true if the receive is successful or false in case of an error. The failure will occur for the following reasons:

- if the handle is invalid
- if the pointer to the receive buffer is NULL
- if the receive size is 0

Remarks

- This function is thread safe in a RTOS application.
- This function should not be called from an interrupt context.

Preconditions

- [DRV_SPI_Open](#) must have been called to obtain a valid opened device handle.
- [DRV_SPI_TransferSetup](#) must have been called if GPIO pin has to be used for chip select or any of the setup parameters has to be changed dynamically.

Example

```
MY_APP_OBJ myAppObj;
uint8_t myRxBuffer[MY_RX_BUFFER_SIZE];

// mySPIHandle is the handle returned by the DRV_SPI_Open function.

if (DRV_SPI_ReadTransfer(mySPIHandle, myRxBuffer, MY_RX_BUFFER_SIZE) == false)
{
    // Handle error here
}
```

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_SPI_Open function.
*pReceiveData	Pointer to the buffer where the data is to be received. For 9 to 15bit mode, data should be right aligned in the 16 bit memory location.
rxSize	Number of bytes to be received. Always, size should be given in terms of bytes. For example, if 10 15-bit data are to be received, the receive size should be 20 bytes.

Function

```
void DRV_SPI_ReadTransfer
(
```

```
const DRV_HANDLE handle,
void* pReceiveData,
size_t rxSize
);
```

DRV_SPI_WriteTransfer Function

This is a blocking function that transmits data over SPI.

File

[drv_spi.h](#)

C

```
bool DRV_SPI_WriteTransfer(const DRV_HANDLE handle, void* pTransmitData, size_t txSize);
```

Returns

- true - transfer is successful
- false - error has occurred

Description

This function does a blocking write operation. The function blocks till the data transmit is complete. Function will return true if the transmit is successful or false in case of an error. The failure will occur for the following reasons:

- if the handle is invalid
- if the pointer to the transmit buffer is NULL
- if the transmit size is 0

Remarks

- This function is thread safe in a RTOS application.
- This function should not be called from an interrupt context.

Preconditions

- [DRV_SPI_Open](#) must have been called to obtain a valid opened device handle.
- [DRV_SPI_TransferSetup](#) must have been called if GPIO pin has to be used for chip select or any of the setup parameters has to be changed dynamically.

Example

```
MY_APP_OBJ myAppObj;
uint8_t myTxBuffer[MY_TX_BUFFER_SIZE];

// mySPIHandle is the handle returned by the DRV_SPI_Open function.

if (DRV_SPI_WriteTransfer(mySPIHandle, myTxBuffer, MY_TX_BUFFER_SIZE) == false)
{
    // Handle error here
}
```

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_SPI_Open function.
*pTransmitData	Pointer to the data which has to be transmitted. For 9 to 15bit mode, data should be right aligned in the 16 bit memory location.
txSize	Number of bytes to be transmitted. Always, size should be given in terms of bytes. For example, if 10 15-bit data are to be transmitted, the transmit size should be 20 bytes.

Function

[void DRV_SPI_WriteTransfer](#)

```

(
const  DRV_HANDLE handle,
void* pTransmitData,
size_t txSize
);

```

DRV_SPI_WriteReadTransfer Function

This is a blocking function that transmits and receives data over SPI.

File

[drv_spi.h](#)

C

```
bool DRV_SPI_WriteReadTransfer(const DRV_HANDLE handle, void* pTransmitData, size_t txSize,
void* pReceiveData, size_t rxSize);
```

Returns

- true - write-read is successful
- false - error has occurred

Description

This function does a blocking write-read operation. The function blocks till the data receive is complete. Function will return true if the receive is successful or false in case of an error. The failure will occur for the following reasons:

- if the handle is invalid
- if the transmit size is non-zero and pointer to the transmit buffer is NULL
- if the receive size is non-zero and pointer to the receive buffer is NULL

Remarks

- This function is thread safe in a RTOS application.
- This function should not be called from an interrupt context.

Preconditions

- [DRV_SPI_Open](#) must have been called to obtain a valid opened device handle.
- [DRV_SPI_TransferSetup](#) must have been called if GPIO pin has to be used for chip select or any of the setup parameters has to be changed dynamically.

Example

```

MY_APP_OBJ myAppObj;
uint8_t myTxBuffer[MY_TX_BUFFER_SIZE];
uint8_t myRxBuffer[MY_RX_BUFFER_SIZE];

// mySPIHandle is the handle returned by the DRV_SPI_Open function.

if (DRV_SPI_WriteReadTransfer(mySPIHandle, myTxBuffer, MY_TX_BUFFER_SIZE,
                             myRxBuffer, MY_RX_BUFFER_SIZE) == false)
{
    // Handle error here
}

```

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_SPI_Open function.
*pTransmitData	Pointer to the data which has to be transmitted. For 9 to 15bit mode, data should be right aligned in the 16 bit memory location.

txSize	Number of bytes to be transmitted. Always, size should be given in terms of bytes. For example, if 10 15-bit data are to be transmitted, the transmit size should be 20 bytes.
*pReceiveData	Pointer to the buffer where the data is to be received. For 9 to 15bit mode, data should be right aligned in the 16 bit memory location.
rxSize	Number of bytes to be received. Always, size should be given in terms of bytes. For example, if 10 15-bit data are to be received, the receive size should be 20 bytes. If "n" number of bytes has to be received AFTER transmitting "m" number of bytes, then "txSize" should be set as "m" and "rxSize" should be set as "m+n".

Function

```
void DRV_SPI_WriteReadTransfer
(
const  DRV_HANDLE handle,
void*  pTransmitData,
size_t  txSize,
void*  pReceiveData,
size_t  rxSize
);
```

DRV_SPI_ReadTransferAdd Function

Queues a read operation.

File

[drv_spi.h](#)

C

```
void DRV_SPI_ReadTransferAdd(const DRV_HANDLE handle, void* pReceiveData, size_t rxSize,
DRV_SPI_TRANSFER_HANDLE * const transferHandle);
```

Returns

None.

Description

This function schedules a non-blocking read operation. The function returns with a valid transfer handle in the transferHandle argument if the request was scheduled successfully. The function adds the request to the instance specific software queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. This API will write rxSize bytes of dummy data and will read rxSize bytes of data in the memory location pointed by pReceiveData.

The function returns [DRV_SPI_TRANSFER_HANDLE_INVALID](#) in the transferHandle argument:

- if pReceiveData is NULL.
- if rxSize is zero.
- if the transfer handle is NULL.
- 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_SPI_TRANSFER_EVENT_COMPLETE](#) event if the transfer was processed successfully or [DRV_SPI_TRANSFER_EVENT_ERROR](#) event if the transfer was not processed successfully.

Remarks

- This function can be called from within the SPI Driver Transfer Event Handler that is registered by the client.
- It should not be called in the event handler associated with another SPI driver instance or event handler of any other peripheral.
- It should not be called directly in any ISR.

Preconditions

- `DRV_SPI_Open` must have been called to obtain a valid opened device handle.
- `DRV_SPI_TransferSetup` must have been called if GPIO pin has to be used for chip select or any of the setup parameters has to be changed dynamically.

Example

```

MY_APP_OBJ myAppObj;
uint8_t myRxBuffer[MY_RX_BUFFER_SIZE];
DRV_SPI_TRANSFER_HANDLE transferHandle;

// mySPIHandle is the handle returned by the DRV_SPI_Open function.

DRV_SPI_ReadTransferAdd(mySPIHandle, myRxBuffer, MY_RX_BUFFER_SIZE, &transferHandle);

if(transferHandle == DRV_SPI_TRANSFER_HANDLE_INVALID)
{
    // Error handling here
}

// Event is received when the transfer is processed.

```

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the <code>DRV_SPI_Open</code> function.
*pReceiveData	Pointer to the location where received data has to be stored. It is user's responsibility to ensure pointed location has sufficient memory to store the read data. For 9 to 15bit mode, received data will be right aligned in the 16 bit memory location.
rxSize	Number of bytes to be received. Always, size should be given in terms of bytes. For example, if 10 15-bit data are to be received, the receive size should be 20 bytes.
transferHandle	Handle which is returned by transfer add function.

Function

```

void DRV_SPI_ReadTransferAdd
(
    const DRV_HANDLE handle,
    void* pReceiveData,
    size_t rxSize,
    DRV_SPI_TRANSFER_HANDLE * const transferHandle
);

```

DRV_SPI_WriteTransferAdd Function

Queues a write operation.

File

`drv_spi.h`

C

```

void DRV_SPI_WriteTransferAdd(const DRV_HANDLE handle, void* pTransmitData, size_t txSize,
DRV_SPI_TRANSFER_HANDLE * const transferHandle);

```

Returns

None.

Description

This function schedules a non-blocking write operation. The function returns with a valid transfer handle in the transferHandle

argument if the request was scheduled successfully. The function adds the request to the instance specific software queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. This API will write txSize bytes of data and the dummy data received will be ignored.

The function returns `DRV_SPI_TRANSFER_HANDLE_INVALID` in the transferHandle argument:

- if pTransmitData is NULL.
- if txSize is zero.
- if the transfer handle is NULL.
- 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_SPI_TRANSFER_EVENT_COMPLETE` event if the transfer was processed successfully or `DRV_SPI_TRANSFER_EVENT_ERROR` event if the transfer was not processed successfully.

Remarks

- This function can be called from within the SPI Driver Transfer Event Handler that is registered by the client.
- It should NOT be called in the event handler associated with another SPI driver instance or event handler of any other peripheral.
- It should not be called directly in any ISR.

Preconditions

- `DRV_SPI_Open` must have been called to obtain a valid opened device handle.
- `DRV_SPI_TransferSetup` must have been called if GPIO pin has to be used for chip select or any of the setup parameters has to be changed dynamically.

Example

```

MY_APP_OBJ myAppObj;
uint8_t myTxBuffer[MY_TX_BUFFER_SIZE];
DRV_SPI_TRANSFER_HANDLE transferHandle;

// mySPIHandle is the handle returned by the DRV_SPI_Open function.

DRV_SPI_WriteTransferAdd(mySPIHandle, myTxBuffer, MY_TX_BUFFER_SIZE, &transferHandle);

if(transferHandle == DRV_SPI_TRANSFER_HANDLE_INVALID)
{
    // Error handling here
}

// Event is received when the transfer is processed.

```

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the <code>DRV_SPI_Open</code> function.
*pTransmitData	Pointer to the data which has to be transmitted. For 9 to 15bit mode, data should be right aligned in the 16 bit memory location.
txSize	Number of bytes to be transmitted. Always, size should be given in terms of bytes. For example, if 10 15-bit data are to be transmitted, the transmit size should be 20 bytes.
transferHandle	Handle which is returned by transfer add function.

Function

```

void DRV_SPI_WriteTransferAdd
(
    const DRV_HANDLE handle,
    void* pTransmitData,
    size_t txSize,
    DRV_SPI_TRANSFER_HANDLE * const transferHandle
);

```

DRV_SPI_WriteReadTransferAdd Function

Queues a write-read transfer operation.

File

[drv_spi.h](#)

C

```
void DRV_SPI_WriteReadTransferAdd(const DRV_HANDLE handle, void* pTransmitData, size_t txSize,
void* pReceiveData, size_t rxSize, DRV_SPI_TRANSFER_HANDLE * const transferHandle);
```

Returns

None.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid transfer handle in the transferHandle argument if the request was scheduled successfully. The function adds the request to the instance specific software queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. This API will write txSize and at the same time counting of rxSize to be read will start. If user wants 'n' bytes to be read after txSize has been written, then he should keep rxSize value as 'txSize + n'.

The function returns [DRV_SPI_TRANSFER_HANDLE_INVALID](#) in the transferHandle argument:

- if neither of the transmit or receive arguments are valid.
- if the transfer handle is NULL.
- 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_SPI_TRANSFER_EVENT_COMPLETE](#) event if the transfer was processed successfully or [DRV_SPI_TRANSFER_EVENT_ERROR](#) event if the transfer was not processed successfully.

Remarks

- This function can be called from within the SPI Driver Transfer Event Handler that is registered by the client.
- It should not be called in the event handler associated with another SPI driver instance or event handler of any other peripheral.
- It should not be called directly in any ISR.

Preconditions

- [DRV_SPI_Open](#) must have been called to obtain a valid opened device handle.
- [DRV_SPI_TransferSetup](#) must have been called if GPIO pin has to be used for chip select or any of the setup parameters has to be changed dynamically.

Example

```
MY_APP_OBJ myAppObj;
uint8_t myTxBuffer[MY_TX_BUFFER_SIZE];
uint8_t myRxBuffer[MY_RX_BUFFER_SIZE];
DRV_SPI_TRANSFER_HANDLE transferHandle;

// mySPIHandle is the handle returned by the DRV_SPI_Open function.

DRV_SPI_WriteReadTransferAdd(mySPIHandle, myTxBuffer, MY_TX_BUFFER_SIZE,
                             myRxBuffer, MY_RX_BUFFER_SIZE, &transferHandle);

if(transferHandle == DRV_SPI_TRANSFER_HANDLE_INVALID)
{
    // Error handling here
}

// Event is received when the transfer is processed.
```

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_SPI_Open function.
*pTransmitData	Pointer to the data which has to be transmitted. if it is NULL, that means only data receiving is expected. For 9 to 15bit mode, data should be right aligned in the 16 bit memory location.
txSize	Number of bytes to be transmitted. Always, size should be given in terms of bytes. For example, if 10 15-bit data are to be transmitted, the transmit size should be 20 bytes.
*pReceiveData	Pointer to the location where received data has to be stored. It is user's responsibility to ensure pointed location has sufficient memory to store the read data. if it is NULL, that means only data transmission is expected. For 9 to 15bit mode, received data will be right aligned in the 16 bit memory location.
rxSize	Number of bytes to be received. Always, size should be given in terms of bytes. For example, if 10 15-bit data are to be received, the receive size should be 20 bytes. If "n" number of bytes has to be received AFTER transmitting "m" number of bytes, then "txSize" should be set as "m" and "rxSize" should be set as "m+n".
transferHandle	Handle which is returned by transfer add function.

Function

```
void DRV_SPI_WriteReadTransferAdd
(
    const    DRV_HANDLE handle,
    void*    pTransmitData,
    size_t    txSize,
    void*    pReceiveData,
    size_t    rxSize,
    DRV_SPI_TRANSFER_HANDLE * const transferHandle
);
```

d) Transfer Status Functions

DRV_SPI_TransferStatusGet Function

Returns transfer add request status.

File

[drv_spi.h](#)

C

```
DRV_SPI_TRANSFER_EVENT DRV_SPI_TransferStatusGet(const DRV_SPI_TRANSFER_HANDLE transferHandle);
```

Returns

One of the elements of the enum "[DRV_SPI_TRANSFER_EVENT](#)".

Description

This function can be used to poll the status of the queued transfer request if the application doesn't prefer to use the event handler (callback) function to get notified.

Remarks

None.

Preconditions

Either `DRV_SPI_ReadTransferAdd` or `DRV_SPI_WriteTransferAdd` or `DRV_SPI_WriteReadTransferAdd` function must have been called and a valid transfer handle must have been returned.

Example

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

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_SPI_TRANSFER_HANDLE transferHandle;
DRV_SPI_TRANSFER_EVENT event;

// mySPIHandle is the handle returned by the DRV_SPI_Open function.

DRV_SPI_ReadTransferAdd( mySPIHandle, myBuffer, MY_RECEIVE_SIZE, &transferHandle);

if(transferHandle == DRV_SPI_TRANSFER_HANDLE_INVALID)
{
    // Error handling here
}

//Check the status of the transfer request
//This call can be used to wait until the transfer is completed.
event = DRV_SPI_TransferStatusGet(transferHandle);
```

Parameters

Parameters	Description
transferHandle	Handle of the transfer request of which status has to be obtained.

Function

`DRV_SPI_TRANSFER_EVENT DRV_SPI_TransferStatusGet(const DRV_SPI_TRANSFER_HANDLE transferHandle)`

e) Data Types and Constants

DRV_SPI_TRANSFER_EVENT Enumeration

Identifies the possible events that can result from a transfer add request.

File

`drv_spi.h`

C

```
typedef enum {
    DRV_SPI_TRANSFER_EVENT_PENDING,
    DRV_SPI_TRANSFER_EVENT_COMPLETE,
    DRV_SPI_TRANSFER_EVENT_HANDLE_EXPIRED,
    DRV_SPI_TRANSFER_EVENT_ERROR,
    DRV_SPI_TRANSFER_EVENT_HANDLE_INVALID
} DRV_SPI_TRANSFER_EVENT;
```

Members

Members	Description
DRV_SPI_TRANSFER_EVENT_PENDING	Transfer request is pending
DRV_SPI_TRANSFER_EVENT_COMPLETE	All data were transferred successfully.

DRV_SPI_TRANSFER_EVENT_HANDLE_EXPIRED	Transfer Handle given is expired. It means transfer is completed but with or without error is not known. In case of Non-DMA transfer, since there is no possibility of error, it can be assumed same as DRV_SPI_TRANSFER_EVENT_COMPLETE
DRV_SPI_TRANSFER_EVENT_ERROR	There was an error while processing transfer request.
DRV_SPI_TRANSFER_EVENT_HANDLE_INVALID	Transfer Handle given is invalid

Description

SPI Driver Transfer Events

This enumeration identifies the possible events that can result from a transfer add request caused by the client calling either [DRV_SPI_ReadTransferAdd](#) or [DRV_SPI_WriteTransferAdd](#) or [DRV_SPI_WriteReadTransferAdd](#) functions.

Remarks

Either DRV_SPI_TRANSFER_EVENT_COMPLETE or DRV_SPI_TRANSFER_EVENT_ERROR is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the [DRV_SPI_TransferEventHandlerSet](#) function when a transfer request is completed.

When status polling is used, any one of these events is returned by [DRV_SPI_TransferStatusGet](#) function.

DRV_SPI_TRANSFER_HANDLE Type

Handle identifying the transfer request queued.

File

[drv_spi.h](#)

C

```
typedef uintptr_t DRV_SPI_TRANSFER_HANDLE;
```

Description

SPI Driver Transfer Handle

A transfer handle value is returned by a call to the [DRV_SPI_ReadTransferAdd](#) or [DRV_SPI_WriteTransferAdd](#) or [DRV_SPI_WriteReadTransferAdd](#) functions. This handle is associated with the transfer request passed into the function and it allows the application to track the completion of the transfer request. The transfer handle value returned from the "transfer add" function is returned back to the client by the "event handler callback" function registered with the driver.

This handle can also be used to poll the transfer completion status using [DRV_SPI_TransferStatusGet](#) API.

The transfer handle assigned to a client request expires when a new transfer request is made after the completion of the current request.

Remarks

None

DRV_SPI_TRANSFER_EVENT_HANDLER Type

Pointer to a SPI Driver Transfer Event handler function

File

[drv_spi.h](#)

C

```
typedef void (* DRV_SPI_TRANSFER_EVENT_HANDLER)(DRV_SPI_TRANSFER_EVENT event,  
DRV_SPI_TRANSFER_HANDLE transferHandle, uintptr_t context);
```

Returns

None.

Description

SPI Driver Transfer Event Handler Function Pointer

This data type defines the required function signature for the SPI driver transfer event handling callback function. A client must register a pointer using the transfer event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive transfer 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_SPI_TRANSFER_EVENT_COMPLETE, it means that the data was transferred successfully.
- If the event is DRV_SPI_TRANSFER_EVENT_ERROR, it means that the data was not transferred successfully.
- The transferHandle parameter contains the transfer handle of the transfer request that is 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_SPI_TransferEventHandlerSet](#) 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) of the client that made the transfer add request.
- The event handler function executes in interrupt context of the peripheral. Hence it is recommended of the application to not perform process intensive or blocking operations with in this function.
- The [DRV_SPI_ReadTransferAdd](#), [DRV_SPI_WriteTransferAdd](#) and [DRV_SPI_WriteReadTransferAdd](#) functions can be called in the event handler to add a transfer request to the driver queue. These functions can only be called to add transfers to the driver instance whose event handler is running. For example, SPI2 driver transfer requests cannot be added in SPI1 driver event handler. Similarly, SPIx transfer requests should not be added in event handler of any other peripheral.

Example

```
void APP_MyTransferEventHandler( DRV_SPI_TRANSFER_EVENT event,
                                DRV_SPI_TRANSFER_HANDLE transferHandle,
                                uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SPI_TRANSFER_EVENT_COMPLETE:
            // Handle the completed transfer.
            break;

        case DRV_SPI_TRANSFER_EVENT_ERROR:
            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
transferHandle	Handle identifying the transfer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_SPI_TRANSFER_HANDLE_INVALID Macro

Definition of an invalid transfer handle.

File

[drv_spi.h](#)

C

```
#define DRV_SPI_TRANSFER_HANDLE_INVALID
```

Description

SPI Driver Invalid Transfer Handle

This is the definition of an invalid transfer handle. An invalid transfer handle is returned by [DRV_SPI_WriteReadTransferAdd](#) or [DRV_SPI_WriteTransferAdd](#) or [DRV_SPI_ReadTransferAdd](#) function if the buffer add request was not successful. It can happen due to invalid arguments or lack of space in the queue.

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.

SYS_MODULE_INDEX Type

Identifies which instance of a system module should be initialized or opened.

File

[system_module.h](#)

C

```
typedef unsigned short int SYS_MODULE_INDEX;
```

Description

System Module Index

This data type identifies to which instance of a system module a call to that module's "Initialize" and "Open" routines refers.

Remarks

Each individual module will usually define macro names for the index values it supports (e.g., DRV_TMR_INDEX_1, DRV_TMR_INDEX_2, ...).

SYS_MODULE_OBJ Type

Handle to an instance of a system module.

File

[system_module.h](#)

C

```
typedef uintptr_t SYS_MODULE_OBJ;
```

Description

System Module Object

This data type is a handle to a specific instance of a system module (such as a device driver).

Remarks

Code outside of a specific module should consider this as an opaque type (much like a void *). Do not make any assumptions about base type as it may change in the future or about the value stored in a variable of this type.

SYS_STATUS Enumeration

Identifies the current status/state of a system module (including device drivers).

File

[system_module.h](#)

C

```
typedef enum {
    SYS_STATUS_ERROR_EXTENDED = -10,
    SYS_STATUS_ERROR = -1,
    SYS_STATUS_UNINITIALIZED = 0,
    SYS_STATUS_BUSY = 1,
    SYS_STATUS_READY = 2,
    SYS_STATUS_READY_EXTENDED = 10
} SYS_STATUS;
```

Members

Members	Description
SYS_STATUS_ERROR_EXTENDED = -10	Indicates that a non-system defined error has occurred. The caller must call the extended status routine for the module in question to identify the error.
SYS_STATUS_ERROR = -1	An unspecified error has occurred.
SYS_STATUS_UNINITIALIZED = 0	The module has not yet been initialized
SYS_STATUS_BUSY = 1	An operation is currently in progress
SYS_STATUS_READY = 2	Any previous operations have succeeded and the module is ready for additional operations
SYS_STATUS_READY_EXTENDED = 10	Indicates that the module is in a non-system defined ready/run state. The caller must call the extended status routine for the module in question to identify the state.

Description

System Module Status

This enumeration identifies the current status/state of a system module (including device drivers).

Remarks

This enumeration is the return type for the system-level status routine defined by each device driver or system module (for example, [DRV_I2C_Status](#)).

SYS_MODULE_OBJ_INVALID Macro

Object handle value returned if unable to initialize the requested instance of a system module.

File

[system_module.h](#)

C

```
#define SYS_MODULE_OBJ_INVALID ((SYS_MODULE_OBJ) -1)
```

Description

System Module Object Invalid

This is the object handle value returned if unable to initialize the requested instance of a system module.

Remarks

Do not rely on the actual value of this constant. It may change in future implementations.

MAIN_RETURN Macro

Defines the correct return type for the "main" routine.

File

[system_common.h](#)

C

```
#define MAIN_RETURN int
```

Description

Main Function Return Type

This macro defines the correct return type for the "main" routine for the selected Microchip microcontroller family.

Remarks

The main function return type may change, depending upon which family of Microchip microcontrollers is chosen. Refer to the user documentation for the C-language compiler in use for more information.

Example

```
MAIN_RETURN main ( void )
{
    // Initialize the system
    SYS_Initialize(...);

    // Main Loop
    while(true)
    {
        SYS_Tasks();
    }

    return MAIN_RETURN_CODE(MAIN_RETURN_SUCCESS);
}
```

MAIN_RETURN_CODE Macro

Casts the given value to the correct type for the return code from "main".

File

[system_common.h](#)

C

```
#define MAIN_RETURN_CODE(c) ((MAIN_RETURN)(c))
```

Description

Main Routine Code Macro

This macro cases the given value to the correct type for the return code from the main function.

Remarks

The main function return type may change, depending upon which family of Microchip microcontrollers is chosen. Refer to the user documentation for the C-language compiler in use for more information.

Example

```
MAIN_RETURN main ( void )
{
    // Initialize the system
    SYS_Initialize(...);
```

```

// Main Loop
while(true)
{
    SYS_Tasks();
}

return MAIN_RETURN_CODE(MAIN_RETURN_SUCCESS);
}

```

SYS_ASSERT Macro

Implements default system assert routine, asserts that "test" is true.

File

[system_common.h](#)

C

```
#define SYS_ASSERT(test,message)
```

Returns

None. Normally hangs in a loop, depending on the implementation to which it is mapped.

Description

This macro implements the default system assert routine that asserts that the provided boolean test is true.

Remarks

Can be overridden as desired by defining your own SYS_ASSERT macro before including [system.h](#).

The default definition removes this macro from all code because it adds significant size to all projects. The most efficient use is to enable it in individual libraries and build them separately for debugging.

Preconditions

None, depending on the implementation to which this macro is mapped.

Example

```

void MyFunc ( int *pointer )
{
    SYS_ASSERT(NULL != pointer, "NULL Pointer passed to MyFunc");

    Do something with pointer.
}

```

Parameters

Parameters	Description
test	This is an expression that resolves to a boolean value (zero=false, non-zero=true)
message	This is a NULL-terminated ASCII character string that can be displayed on a debug output terminal if "test" is false (if supported).

Function

`void SYS_ASSERT (bool test, char *message)`

SYS_MODULE_DEINITIALIZE_ROUTINE Type

Pointer to a routine that deinitializes a system module (driver, library, or system-maintained application).

File

[system_module.h](#)

C

```
typedef void (* SYS_MODULE_DEINITIALIZE_ROUTINE)(SYS_MODULE_OBJ object);
```

Returns

None.

Description

System Module Deinitialization Routine Pointer. This data type is a pointer to a routine that deinitializes a system module (driver, library, or system-maintained application).

Remarks

If the module instance has to be used again, the module's "initialize" function must first be called.

Preconditions

The low-level board initialization must have (and will be) completed and the module's initialization routine will have been called before the system will call the deinitialization routine for any modules.

Example

None.

Parameters

Parameters	Description
object	Handle to the module instance

Function

```
void (* SYS_MODULE_DEINITIALIZE_ROUTINE)( SYS_MODULE_OBJ object )
```

SYS_MODULE_INITIALIZE_ROUTINE Type

Pointer to a routine that initializes a system module (driver, library, or system-maintained application).

File

[system_module.h](#)

C

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

Returns

A handle to the instance of the system module that was initialized. This handle is a necessary parameter to all of the other system-module level routines for that module.

Description

System Module Initialization Routine Pointer

This data type is a pointer to a routine that initializes a system module (driver, library, or system-maintained application).

Remarks

This function will only be called once during system initialization.

Preconditions

The low-level board initialization must have (and will be) completed before the system will call the initialization routine for any modules.

Parameters

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

Function

```
SYS_MODULE_OBJ (* SYS_MODULE_INITIALIZE_ROUTINE) (
  const SYS_MODULE_INDEX index,
  const SYS_MODULE_INIT * const init )
```

SYS_MODULE_REINITIALIZE_ROUTINE Type

Pointer to a routine that reinitializes a system module (driver, library, or system-maintained application)

File

[system_module.h](#)

C

```
typedef void (* SYS_MODULE_REINITIALIZE_ROUTINE)(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

None.

Description

System Module Reinitialization Routine Pointer

This data type is a pointer to a routine that reinitializes a system module (driver, library, or system-maintained application).

Remarks

This operation uses the same initialization data structure as the Initialize operation.

This operation can be used to refresh the hardware state as defined by the initialization data, thus it must guarantee that all hardware state has been refreshed.

This function can be called multiple times to reinitialize the module.

Preconditions

The low-level board initialization must have (and will be) completed and the module's initialization routine will have been called before the system will call the reinitialization routine for any modules.

Example

None.

Parameters

Parameters	Description
object	Handle to the module instance
init	Pointer to the data structure containing any data necessary to initialize the module. This pointer may be null if no data is required and default initialization is to be used.

Function

```
void (* SYS_MODULE_REINITIALIZE_ROUTINE) ( SYS_MODULE_OBJ object,
  const SYS_MODULE_INIT * const init)
```

SYS_MODULE_STATUS_ROUTINE Type

Pointer to a routine that gets the current status of a system module (driver, library, or system-maintained application).

File

[system_module.h](#)

C

```
typedef SYS_STATUS (* SYS_MODULE_STATUS_ROUTINE)(SYS_MODULE_OBJ object);
```

Returns

One of the possible status codes from [SYS_STATUS](#)

Description

System Module Status Routine Pointer

This data type is a pointer to a routine that gets the current status of a system module (driver, library, or system-maintained application).

Remarks

A module's status operation can be used to determine when any of the other module level operations has completed as well as to obtain general status of the module. The value returned by the status routine will be checked after calling any of the module operations to find out when they have 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). A module may define module-specific error values of less or equal SYS_STATUS_ERROR_EXTENDED (-10).

The status function must NEVER block.

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 low-level board initialization must have (and will be) completed and the module's initialization routine will have been called before the system will call the status routine for any modules.

Example

None.

Parameters

Parameters	Description
object	Handle to the module instance

Function

```
SYS_STATUS (* SYS_MODULE_STATUS_ROUTINE)( SYS_MODULE_OBJ object )
```

SYS_MODULE_TASKS_ROUTINE Type

Pointer to a routine that performs the tasks necessary to maintain a state machine in a module system module (driver, library, or system-maintained application).

File

[system_module.h](#)

C

```
typedef void (* SYS_MODULE_TASKS_ROUTINE)(SYS_MODULE_OBJ object);
```

Returns

None.

Description

System Module Tasks Routine Pointer

This data type is a pointer to a routine that performs the tasks necessary to maintain a state machine in a module system module (driver, library, or system-maintained application).

Remarks

If the module is interrupt driven, the system will call this routine from an interrupt context.

Preconditions

The low-level board initialization must have (and will be) completed and the module's initialization routine will have been called before the system will call the deinitialization routine for any modules.

Example

None.

Parameters

Parameters	Description
object	Handle to the module instance

Function

```
void (* SYS_MODULE_TASKS_ROUTINE) ( SYS\_MODULE\_OBJ object )
```

SYS_MODULE_OBJ_STATIC Macro

Object handle value returned by static modules.

File

[system_module.h](#)

C

```
#define SYS_MODULE_OBJ_STATIC ((SYS_MODULE_OBJ) 0 )
```

Description

System Module Object Static

This is the object handle value returned by static system modules.

Remarks

Do not rely on the actual value of this constant. It may change in future implementations.

Files

Files

Name	Description
drv_spi.h	SPI Driver Interface Header File

Description

This section will list only the library's interface header file(s).

drv_spi.h

SPI Driver Interface Header File

Enumerations

	Name	Description
	DRV_SPI_TRANSFER_EVENT	Identifies the possible events that can result from a transfer add request.

Functions

	Name	Description
≡	DRV_SPI_Close	Closes an opened-instance of the SPI driver.
≡	DRV_SPI_Initialize	Initializes the SPI instance for the specified driver index.
≡	DRV_SPI_Open	Opens the specified SPI driver instance and returns a handle to it.
≡	DRV_SPI_ReadTransfer	This is a blocking function that receives data over SPI.
≡	DRV_SPI_ReadTransferAdd	Queues a read operation.
≡	DRV_SPI_Status	Gets the current status of the SPI driver module.
≡	DRV_SPI_TransferEventHandlerSet	Allows a client to set a transfer event handling function for the driver to call back when queued transfer has finished.
≡	DRV_SPI_TransferSetup	Sets the dynamic configuration of the driver including chip select pin.
≡	DRV_SPI_TransferStatusGet	Returns transfer add request status.
≡	DRV_SPI_WriteReadTransfer	This is a blocking function that transmits and receives data over SPI.
≡	DRV_SPI_WriteReadTransferAdd	Queues a write-read transfer operation.
≡	DRV_SPI_WriteTransfer	This is a blocking function that transmits data over SPI.
≡	DRV_SPI_WriteTransferAdd	Queues a write operation.

Macros

	Name	Description
	DRV_SPI_TRANSFER_HANDLE_INVALID	Definition of an invalid transfer handle.

Types

	Name	Description
	DRV_SPI_TRANSFER_EVENT_HANDLER	Pointer to a SPI Driver Transfer Event handler function
	DRV_SPI_TRANSFER_HANDLE	Handle identifying the transfer request queued.

Description

SPI Driver Interface Header File

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

File Name

drv_spi.h

Company

Microchip Technology Inc.

SST26 Driver Library Help

This section describes the SST26 QSPI Flash Memory Driver Library.

Introduction

This library provides an interface to access the SST26 Flash Memory over the QSPI interface.

Description

This library provides a non-blocking interface to read, write and erase SST26 flash memory. The library uses the QSPI PLIB to interface with the SST26 flash.

Key Features:

- Supports Multiple variants of the SST26 Flash devices.
- Supports a single instance of the SST26 Flash and a single client to the driver.
- Supports Sector/Bulk/Chip Erase Operations.
- Supports writes to random memory address within page boundaries.
- The library can be used in both bare-metal and RTOS environments.

Using the Library

This topic describes the basic architecture of the SST26 QSPI Flash Memory Library and provides information on how it works.

Description

The SST26 library provides non-blocking APIs to read, write and erase SST26 flash memory. It uses the QSPI peripheral library to interface with the SST26.

- The library can be used to perform page write to SST26 Flash. Here, the memory start address must be aligned to the EEPROM page boundary.
- The library can be used to perform Sector/Bulk/Chip Erase operations.
- The library can be used to Unlock flash before performing Erase/Write operations.
- The library can be used to read flash JEDEC-ID.
- The Client should poll for the status of the data transfer.
- The library can be used in both bare-metal and RTOS environments.
- The library can be used to interface with Memory Driver to perform Block operations on the SST26 Flash.

Abstraction Model

This library provides an abstraction of SST26 Driver Library.

Description

The SST26 library interface provides read, write and Erase functions that abstract out the internal workings of the SST26 driver and the underlying QSPI protocol.

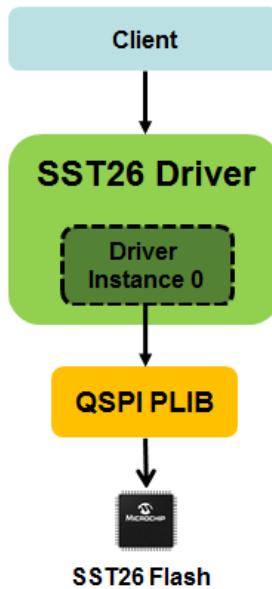


Figure-1 SST26 Driver Block Diagram

How the Library Works

This topic describes the basic architecture of the SST26 Driver Library and provides information on how it works.

Description

- The SST26 Driver Library performs set of flash operation during initialization. If any of the below flash operations fail the Driver will not be ready to use. This status of the driver can be checked using [DRV_SST26_Status\(\)](#).
 - Resets the Flash
 - Configures the flash device to QUAD IO Mode
 - Unlocks the flash
- When the client request for device geometry, it queries for the JEDEC-Id and populates the geometry table [DRV_SST26_GEOMETRY](#) appropriately.
- The library does not support queuing of more than one requests. The application must check and ensure that any previous request is completed before submitting a new one. This can be done by polling the status of data transfer using [DRV_SST26_TransferStatusGet\(\)](#).

Configuring the Library

This Section provides information on how to configure SST26 QSPI Flash Memory Driver library.

Description

SST26 Driver library should be configured via MHC. Below are the Snapshot of the MHC configuration window for SST26 driver and brief description.

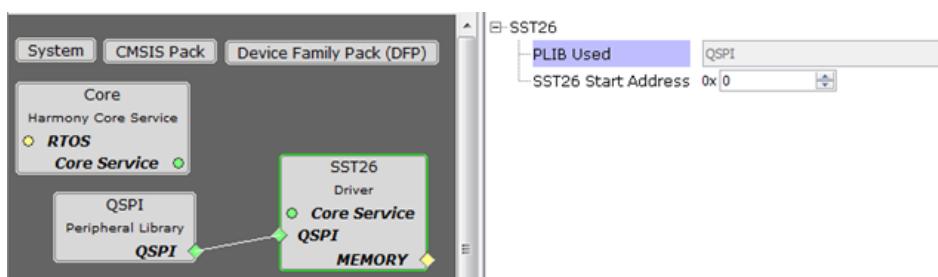


Figure-2 SST26 Driver without Connecting to Memory Driver

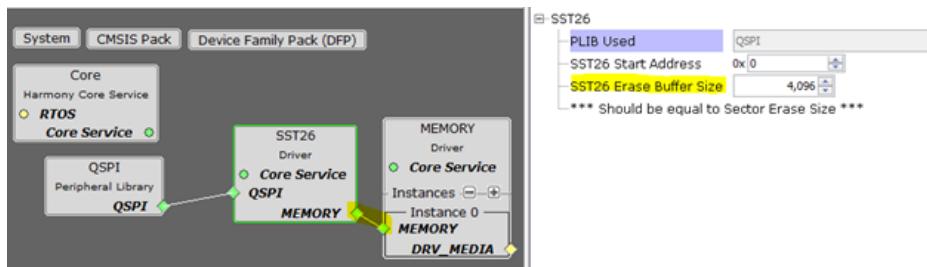


Figure-3 SST26 Driver with Connection to Memory Driver

Configuration Options:

1. PLIB Used

1. Specifies the Peripheral library connected

2. SST26 Start Address

1. Specifies the flash memory start address to be used for Transfer operations.
2. The start address will be populated in the device geometry table [DRV_SST26_GEOMETRY](#)

3. SST26 Erase Buffer Size

1. Specifies the size for erase buffer used by Memory driver.
2. The size of the buffer should be equal to erase sector size as the memory driver will call [DRV_SST26_SectorErase](#).
3. This option appears only when SST26 Driver is connected to Memory Driver for block operations. Refer [Figure-3](#)

Building the Library

This Section provides information on how the SST26 QSPI Flash Memory Library can be built.

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Functions

	Name	Description
≡	DRV_SST26_Initialize	Initializes the SST26 Driver

b) Core Client Functions

	Name	Description
≡	DRV_SST26_Open	Opens the specified SST26 driver instance and returns a handle to it
≡	DRV_SST26_Close	Closes an opened-instance of the SST26 driver

c) Transfer Functions

	Name	Description
≡	DRV_SST26_UnlockFlash	Unlocks the flash device for Erase and Program operations.
≡	DRV_SST26_ReadJedecId	Reads JEDEC-ID of the flash device.
≡	DRV_SST26_SectorErase	Erase the sector from the specified block start address.
≡	DRV_SST26_BulkErase	Erase a block from the specified block start address.
≡	DRV_SST26_ChipErase	Erase entire flash memory.
≡	DRV_SST26_PageWrite	Writes one page of data starting at the specified address.
≡	DRV_SST26_Read	Reads n bytes of data from the specified start address of flash memory.

	DRV_SST26_Status	Gets the current status of the SST26 driver module.
	DRV_SST26_TransferStatusGet	Gets the current status of the transfer request.

d) Block Interface Functions

	Name	Description
	DRV_SST26_GeometryGet	Returns the geometry of the device.

e) Data Types and Constants

	Name	Description
	DRV_SST26_GEOMETRY	SST26 Device Geometry data.
	DRV_SST26_TRANSFER_STATUS	SST26 Driver Transfer Status

Description

This section describes the Application Programming Interface (API) functions of the SST26 QSPI Flash Memory Library.

a) System Functions

DRV_SST26_Initialize Function

Initializes the SST26 Driver

File

[drv_sst26.h](#)

C

```
SYS_MODULE_OBJ DRV_SST26_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 SST26 driver making it ready for client to use.

- Resets the Flash Device
- Puts it on QUAD IO Mode
- Unlocks the flash

Remarks

This routine must be called before any other SST26 driver routine is called.

This routine should only be called once during system initialization.

This routine will block for hardware access.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the SST26 Driver
// with SST26 QSPI flash device attached.
```

```
SYS_MODULE_OBJ objectHandle;
```

```
const DRV_SST26_PLIB_INTERFACE drvsSST26PlibAPI = {
    .CommandWrite = QSPI_CommandWrite,
    .RegisterRead = QSPI_RegisterRead,
```

```

    .RegisterWrite = QSPI_RegisterWrite,
    .MemoryRead   = QSPI_MemoryRead,
    .MemoryWrite   = QSPI_MemoryWrite
};

const DRV_SST26_INIT drvSST26InitData =
{
    .sst26Plib      = &drvSST26PlibAPI,
};

objectHandle = DRV_SST26_Initialize((SYS_MODULE_INDEX)DRV_SST26_INDEX, (SYS_MODULE_INIT *)
*)&drvSST26InitData);

if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}

```

Parameters

Parameters	Description
drvIndex	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_SST26_Initialize
(
const SYS_MODULE_INDEX drvIndex,
const SYS_MODULE_INIT *const init
);

```

b) Core Client Functions

DRV_SST26_Open Function

Opens the specified SST26 driver instance and returns a handle to it

File

[drv_sst26.h](#)

C

```
DRV_HANDLE DRV_SST26_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, [DRV_HANDLE_INVALID](#) is returned. Errors can occur under the following circumstances:

- if the driver hardware instance being opened is not initialized.

Description

This routine opens the specified SST26 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_SST26_Close](#) routine is called.

If the driver has already been opened, it should not be opened again.

Preconditions

Function [DRV_SST26_Initialize](#) must have been called before calling this function.

Driver should be in ready state to accept the request. Can be checked by calling [DRV_SST26_Status\(\)](#).

Example

```
DRV_HANDLE handle;

handle = DRV_SST26_Open(DRV_SST26_INDEX);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the 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_SST26_Open\(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent \);](#)

DRV_SST26_Close Function

Closes an opened-instance of the SST26 driver

File

[drv_sst26.h](#)

C

```
void DRV_SST26_Close(const DRV_HANDLE handle);
```

Returns

None

Description

This routine closes an opened-instance of the SST26 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_SST26_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

[DRV_SST26_Open](#) must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle; // Returned from DRV_SST26_Open

DRV_SST26_Close(handle);
```

Parameters

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

Function

```
void DRV_SST26_Close( const DRV_HANDLE handle );
```

c) Transfer Functions

DRV_SST26_UnlockFlash Function

Unlocks the flash device for Erase and Program operations.

File

[drv_sst26.h](#)

C

```
bool DRV_SST26_UnlockFlash(const DRV_HANDLE handle);
```

Returns

false

- if Write enable fails before sending unlock command to flash
- if Unlock flash command itself fails

true

- if the unlock is successfully completed

Description

This function schedules a blocking operation for unlocking the flash blocks globally. This allows to perform erase and program operations on the flash.

The request is sent in QUAD_MODE to flash device.

Remarks

This routine will block wait for request to complete.

Preconditions

The [DRV_SST26_Open\(\)](#) routine must have been called for the specified SST26 driver instance.

Example

```
DRV_HANDLE handle; // Returned from DRV_SST26_Open

if(true != DRV_SST26_UnlockFlash(handle))
{
    // Error handling here
}
```

Parameters

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

Function

```
bool DRV_SST26_UnlockFlash( const DRV_HANDLE handle );
```

DRV_SST26_ReadJedecId Function

Reads JEDEC-ID of the flash device.

File

[drv_sst26.h](#)

C

```
bool DRV_SST26_ReadJedecId(const DRV_HANDLE handle, void * jedec_id);
```

Returns

false

- if read jedec-id command fails

true

- if the read is successfully completed

Description

This function schedules a blocking operation for reading the JEDEC-ID. This information can be used to get the flash device geometry.

The request is sent in QUAD_MODE to flash device.

Remarks

This routine will block wait for transfer to complete.

Preconditions

The [DRV_SST26_Open\(\)](#) routine must have been called for the specified SST26 driver instance.

Example

```
DRV_HANDLE handle; // Returned from DRV_SST26_Open
uint32_t jedec_id = 0;

if(true != DRV_SST26_ReadJedecId(handle, &jedec_id))
{
    // Error handling here
}
```

Parameters

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

Function

```
bool DRV_SST26_ReadJedecId( const DRV_HANDLE handle, void *jedec_id );
```

DRV_SST26_SectorErase Function

Erase the sector from the specified block start address.

File

[drv_sst26.h](#)

C

```
bool DRV_SST26_SectorErase(const DRV_HANDLE handle, uint32_t address);
```

Returns

false

- if Write enable fails before sending sector erase command to flash
- if sector erase command itself fails

true

- if the erase request is successfully sent to the flash

Description

This function schedules a non-blocking sector erase operation of flash memory. Each Sector is of 4 KByte.

The requesting client should call [DRV_SST26_TransferStatusGet\(\)](#) API to know the current status of the request.

The request is sent in QUAD_MODE to flash device.

Remarks

This routine will block wait until erase request is submitted successfully.

Client should wait until erase is complete to send next transfer request.

Preconditions

The [DRV_SST26_Open\(\)](#) routine must have been called for the specified SST26 driver instance.

Example

```
DRV_HANDLE handle; // Returned from DRV_SST26_Open
uint32_t sectorStart = 0;

if(false == DRV_SST26_SectorErase(handle, sectorStart))
{
    // Error handling here
}

// Wait for erase to be completed
while(DRV_SST26_TRANSFER_BUSY == DRV_SST26_TransferStatusGet(handle));
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
address	block start address from where a sector needs to be erased.

Function

```
bool DRV_SST26_SectorErase( const DRV_HANDLE handle, uint32_t address );
```

DRV_SST26_BulkErase Function

Erase a block from the specified block start address.

File

[drv_sst26.h](#)

C

```
bool DRV_SST26_BulkErase( const DRV_HANDLE handle, uint32_t address );
```

Returns

false

- if Write enable fails before sending sector erase command to flash
- if block erase command itself fails

true

- if the erase request is successfully sent to the flash

Description

This function schedules a non-blocking block erase operation of flash memory. The block size can be 8 KByte, 32KByte or 64 KByte.

The requesting client should call [DRV_SST26_TransferStatusGet\(\)](#) API to know the current status of the request.

The request is sent in QUAD_MODE to flash device.

Remarks

This routine will block wait until erase request is submitted successfully.
Client should wait until erase is complete to send next transfer request.

Preconditions

The [DRV_SST26_Open\(\)](#) routine must have been called for the specified SST26 driver instance.

Example

```
DRV_HANDLE handle; // Returned from DRV_SST26_Open
uint32_t blockStart = 0;

if(false == DRV_SST26_SectorErase(handle, blockStart))
{
    // Error handling here
}

// Wait for erase to be completed
while(DRV_SST26_TRANSFER_BUSY == DRV_SST26_TransferStatusGet(handle));
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
address	block start address to be erased.

Function

bool DRV_SST26_BulkErase(const [DRV_HANDLE](#) handle, uint32_t address);

DRV_SST26_ChipErase Function

Erase entire flash memory.

File

[drv_sst26.h](#)

C

```
bool DRV_SST26_ChipErase( const DRV_HANDLE handle );
```

Returns

false

- if Write enable fails before sending sector erase command to flash
- if chip erase command itself fails

true

- if the erase request is successfully sent to the flash

Description

This function schedules a non-blocking chip erase operation of flash memory.

The requesting client should call [DRV_SST26_TransferStatusGet\(\)](#) API to know the current status of the request.

The request is sent in QUAD_MODE to flash device.

Remarks

This routine will block wait until erase request is submitted successfully.
Client should wait until erase is complete to send next transfer request.

Preconditions

The [DRV_SST26_Open\(\)](#) routine must have been called for the specified SST26 driver instance.

Example

```
DRV_HANDLE handle; // Returned from DRV_SST26_Open

if(false == DRV_SST26_ChipErase(handle))
{
    // Error handling here
}

// Wait for erase to be completed
while(DRV_SST26_TRANSFER_BUSY == DRV_SST26_TransferStatusGet(handle));
```

Parameters

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

Function

```
bool DRV_SST26_ChipErase( const DRV_HANDLE handle );
```

DRV_SST26_PageWrite Function

Writes one page of data starting at the specified address.

File

[drv_sst26.h](#)

C

```
bool DRV_SST26_PageWrite( const DRV_HANDLE handle, void * tx_data, uint32_t address );
```

Returns

false

- if Write enable fails before sending sector erase command to flash
- if write command itself fails

true

- if the write request is successfully sent to the flash

Description

This function schedules a non-blocking write operation for writing maximum one page of data into flash memory.

The requesting client should call [DRV_SST26_TransferStatusGet\(\)](#) API to know the current status of the request.

The request is sent in QUAD_MODE to flash device.

Remarks

This routine will block wait until write request is submitted successfully.

Client should wait until write is complete to send next transfer request.

Preconditions

The [DRV_SST26_Open\(\)](#) routine must have been called for the specified SST26 driver instance.

The flash address location which has to be written, must have been erased before using the [SST26_xxxErase\(\)](#) routine.

The flash address has to be a Page aligned address.

Example

```
#define PAGE_SIZE    256
#define BUFFER_SIZE  1024
#define MEM_ADDRESS  0x0

DRV_HANDLE handle; // Returned from DRV_SST26_Open
uint8_t writeBuffer[BUFFER_SIZE];
```

```

bool status = false;

if(false == DRV_SST26_SectorErase(handle))
{
    // Error handling here
}

// Wait for erase to be completed
while(DRV_SST26_TRANSFER_BUSY == DRV_SST26_TransferStatusGet(handle));

for (uint32_t j = 0; j < BUFFER_SIZE; j += PAGE_SIZE)
{
    if (true != DRV_SST26_PageWrite(handle, (void *)&writeBuffer[j], (MEM_ADDRESS + j)))
    {
        status = false;
        break;
    }

    // Wait for write to be completed
    while(DRV_SST26_TRANSFER_BUSY == DRV_SST26_TransferStatusGet(handle));
    status = true;
}

if(status == false)
{
    // Error handling here
}

```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
*tx_data	The source buffer containing data to be programmed into SST26 Flash
tx_data_length	Total number of bytes to be written. should not be greater than page size
address	Write memory start address from where the data should be written

Function

```
bool DRV_SST26_PageWrite( const DRV_HANDLE handle, void *tx_data, uint32_t tx_data_length, uint32_t address );
```

DRV_SST26_Read Function

Reads n bytes of data from the specified start address of flash memory.

File

[drv_sst26.h](#)

C

```
bool DRV_SST26_Read( const DRV_HANDLE handle, void * rx_data, uint32_t rx_data_length, uint32_t address );
```

Returns

- false
 - if read command itself fails
- true
 - if number of bytes requested are read from flash memory

Description

This function schedules a blocking operation for reading requested number of data bytes from the flash memory.

The request is sent in QUAD_MODE to flash device.

Remarks

This routine will block waiting until read request is completed successfully.

Preconditions

The [DRV_SST26_Open\(\)](#) routine must have been called for the specified SST26 driver instance.

Example

```
#define BUFFER_SIZE 1024
#define MEM_ADDRESS 0x0

DRV_HANDLE handle; // Returned from DRV_SST26_Open
uint8_t readBuffer[BUFFER_SIZE];

if (true != DRV_SST26_Read(handle, (void *)&readBuffer, BUFFER_SIZE, MEM_ADDRESS))
{
    // Error handling here
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
*rx_data	Buffer pointer into which the data read from the SST26 Flash memory will be placed.
rx_data_length	Total number of bytes to be read.
address	Read memory start address from where the data should be read.

Function

```
bool DRV_SST26_Read( const DRV_HANDLE handle, void *rx_data, uint32_t rx_data_length, uint32_t address );
```

DRV_SST26_Status Function

Gets the current status of the SST26 driver module.

File

[drv_sst26.h](#)

C

```
SYS_STATUS DRV_SST26_Status( const SYS_MODULE_INDEX drvIndex );
```

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.

SYS_STATUS_BUSY - Indicates the driver is in busy state.

Description

This routine provides the current status of the SST26 driver module.

Remarks

This routine will NEVER block wait for hardware.

Preconditions

Function [DRV_SST26_Initialize](#) should have been called before calling this function.

Example

```
SYS_STATUS Status;

Status = DRV_SST26_Status(DRV_SST26_INDEX);
```

Parameters

Parameters	Description
drvIndex	Identifier for the instance used to initialize driver

Function

`SYS_STATUS DRV_SST26_Status(const SYS_MODULE_INDEX drvIndex);`

DRV_SST26_TransferStatusGet Function

Gets the current status of the transfer request.

File

`drv_sst26.h`

C

`DRV_SST26_TRANSFER_STATUS DRV_SST26_TransferStatusGet(const DRV_HANDLE handle);`

Returns

`DRV_SST26_TRANSFER_ERROR_UNKNOWN`

- If the flash status register read request fails

`DRV_SST26_TRANSFER_BUSY`

- If the current transfer request is still being processed

`DRV_SST26_TRANSFER_COMPLETED`

- If the transfer request is completed

Description

This routine gets the current status of the transfer request. The application must use this routine where the status of a scheduled request needs to be polled on.

Remarks

This routine will block for hardware access.

Preconditions

The `DRV_SST26_Open()` routine must have been called for the specified SST26 driver instance.

Example

```
DRV_HANDLE handle; // Returned from DRV_SST26_Open

if (DRV_SST26_TRANSFER_COMPLETED == DRV_SST26_TransferStatusGet(handle))
{
    // Operation Done
}
```

Parameters

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

Function

`DRV_SST26_TRANSFER_STATUS DRV_SST26_TransferStatusGet(const DRV_HANDLE handle);`

d) Block Interface Functions

DRV_SST26_GeometryGet Function

Returns the geometry of the device.

File

[drv_sst26.h](#)

C

```
bool DRV_SST26_GeometryGet(const DRV_HANDLE handle, DRV_SST26_GEOMETRY * geometry);
```

Returns

false

- if read device id fails

true

- if able to get the geometry details of the flash

Description

This API gives the following geometrical details of the SST26 Flash:

- Number of Read/Write/Erase Blocks and their size in each region of the device
- Flash block start address.

Remarks

This API is more useful when used to interface with block driver like memory driver.

Preconditions

The [DRV_SST26_Open\(\)](#) routine must have been called for the specified SST26 driver instance.

Example

```
DRV_HANDLE handle; // Returned from DRV_SST26_Open
DRV_SST26_GEOMETRY sst26FlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

DRV_SST26_GeometryGet(handle, &sst26FlashGeometry);

readBlockSize = sst26FlashGeometry.read_blockSize;
nReadBlocks = sst26FlashGeometry.read_numBlocks;
nReadRegions = sst26FlashGeometry.numReadRegions;

writeBlockSize = sst26FlashGeometry.write_blockSize;
eraseBlockSize = sst26FlashGeometry.erase_blockSize;

totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
*geometry_table	pointer to flash device geometry table instance

Function

```
bool DRV_SST26_GeometryGet( const DRV_HANDLE handle, SST26_GEOMETRY *geometry );
```

e) Data Types and Constants

DRV_SST26_GEOMETRY Structure

SST26 Device Geometry data.

File

[drv_sst26.h](#)

C

```
typedef struct {
    uint32_t read_blockSize;
    uint32_t read_numBlocks;
    uint32_t numReadRegions;
    uint32_t write_blockSize;
    uint32_t write_numBlocks;
    uint32_t numWriteRegions;
    uint32_t erase_blockSize;
    uint32_t erase_numBlocks;
    uint32_t numEraseRegions;
    uint32_t blockStartAddress;
} DRV_SST26_GEOMETRY;
```

Description

This data type will be used to get the geometry details of the SST26 flash device.

Remarks

None.

DRV_SST26_TRANSFER_STATUS Enumeration

SST26 Driver Transfer Status

File

[drv_sst26.h](#)

C

```
typedef enum {
    DRV_SST26_TRANSFER_BUSY,
    DRV_SST26_TRANSFER_COMPLETED,
    DRV_SST26_TRANSFER_ERROR_UNKNOWN
} DRV_SST26_TRANSFER_STATUS;
```

Members

Members	Description
DRV_SST26_TRANSFER_BUSY	Transfer is being processed
DRV_SST26_TRANSFER_COMPLETED	Transfer is successfully completed
DRV_SST26_TRANSFER_ERROR_UNKNOWN	Transfer had error

Description

This data type will be used to indicate the current transfer status for SST26 driver.

Remarks

None.

Files

Files

Name	Description
drv_sst26.h	SST26 Driver Interface Definition

Description

This section will list only the library's interface header file(s).

[drv_sst26.h](#)

SST26 Driver Interface Definition

Enumerations

	Name	Description
	DRV_SST26_TRANSFER_STATUS	SST26 Driver Transfer Status

Functions

	Name	Description
	DRV_SST26_BulkErase	Erase a block from the specified block start address.
	DRV_SST26_ChipErase	Erase entire flash memory.
	DRV_SST26_Close	Closes an opened-instance of the SST26 driver
	DRV_SST26_GeometryGet	Returns the geometry of the device.
	DRV_SST26_Initialize	Initializes the SST26 Driver
	DRV_SST26_Open	Opens the specified SST26 driver instance and returns a handle to it
	DRV_SST26_PageWrite	Writes one page of data starting at the specified address.
	DRV_SST26_Read	Reads n bytes of data from the specified start address of flash memory.
	DRV_SST26_ReadJedeclD	Reads JEDEC-ID of the flash device.
	DRV_SST26_SectorErase	Erase the sector from the specified block start address.
	DRV_SST26_Status	Gets the current status of the SST26 driver module.
	DRV_SST26_TransferStatusGet	Gets the current status of the transfer request.
	DRV_SST26_UnlockFlash	Unlocks the flash device for Erase and Program operations.

Structures

	Name	Description
	DRV_SST26_GEOMETRY	SST26 Device Geometry data.

Description

SST26 Driver Interface Definition

The SST26 driver provides a simple interface to manage the SST26VF series of SQI Flash Memory connected to Microchip microcontrollers. This file defines the interface definition for the SST26 driver.

File Name

drv_sst26.h

Company

Microchip Technology Inc.

USART Driver Library Help

This section describes the USART Driver Library.

Introduction

This library provides an interface to manage the data transfer operations using the USART module.

Description

This driver provides application ready routines to read and write data to the USART using common data transfer models, which eliminates the need for the application to implement this code. The USART driver features the following:

- Supports single-client and multi-instance in Asynchronous mode operation
- Supports multi-client and multi-instance in Synchronous mode operation
- Provides data transfer events
- Supports blocking and non-blocking operation
- Features thread-safe functions for use in RTOS applications
- Supports DMA transfers

Using the Library

This topic describes the basic architecture of the USART Driver Library and provides information on how to use it.

Description

The USART driver builds on top of the USART and UART peripheral library (PLIB) and provides write, read and write-read APIs in blocking and non-blocking mode.

- Provides Write and Read APIs.
- Supports only single-client in *asynchronous* mode operation (single-client mode).
- In *asynchronous* (non-blocking) mode, application can either register a callback to get notified once the data transfer is complete or can poll the status of the data transfer using the status related APIs.
- In *asynchronous* mode, application can queue more than one transmit/receive requests without waiting for the previous request to be completed. The number of transmit/receive requests that can be queued depends on the depth of the transfer queue configured using MHC.
- The *asynchronous* mode is supported in both bare-metal and RTOS environment.
- The *synchronous* (blocking) mode of the driver provides a blocking behavior and is supported only in an RTOS environment.
- The *synchronous* mode of the driver does not support callback or queuing multiple requests. This is because the implementation is blocking in nature.
- Supports DMA for Transfer/Receive in both *asynchronous* and *synchronous* mode.

Abstraction Model

The USART Driver Library provides the low-level abstraction of the USART and UART module with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the USART Driver Library interface.

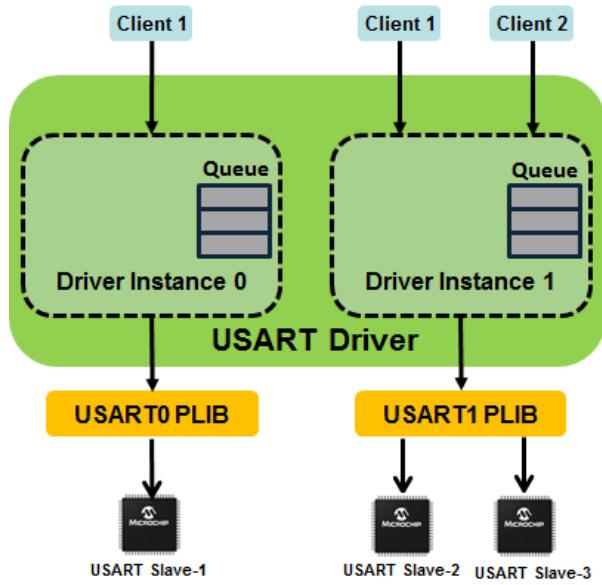
Description

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 system to initialize the driver. The driver must be initialized before it can be opened by a client. 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. 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 USART protocol. The user can use callback mechanisms or use polling to check status of transfer.

The following diagrams illustrate the model used by the USART Driver.



How the Library Works

This topic provides information on how the USART Driver Library works.

Description

Prior to using the USART Driver, the application must configure the USART Driver mode, i.e. whether synchronous (blocking) or asynchronous (non-blocking) mode. Then configurations can be made by selecting each instance.

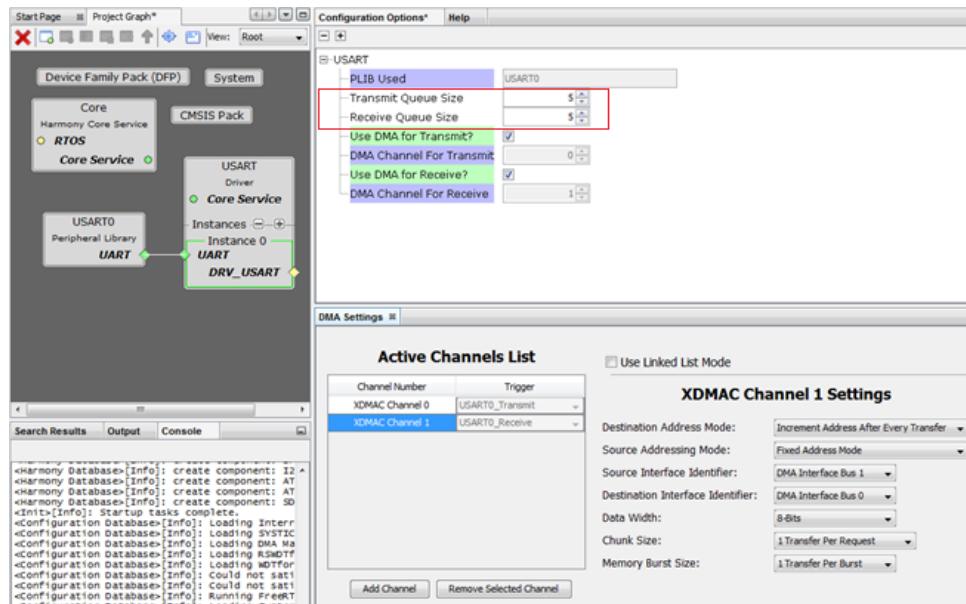
- The USART driver is built on top of the USART/UART peripheral library.
- The USART driver registers a callback with the underlying USART peripheral library to receive transfer related events from the peripheral library. The USART driver callback is called by the peripheral library from the interrupt context.
- The USART driver state machine runs from the interrupt context. Once the transfer is complete, the driver calls the callback registered by the application (from the interrupt context).
- Each instance of the driver (in asynchronous/non-blocking mode) has a dedicated queue which can be configured using the MHC configuration options. The requests submitted by the clients are queued in the respective driver instance request queue.
- The USART driver can be used with DMA for data Transfer/Receive.
- The USART driver is capable of supporting multiple instances of the USART/UART peripheral.

Configuring the Library

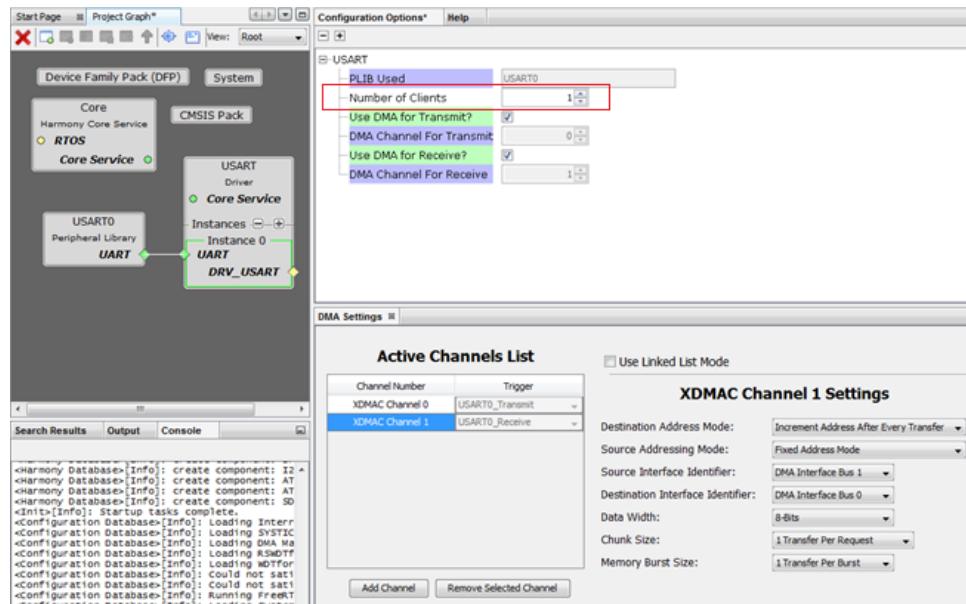
This Section provides information on how to configure the USART Driver library.

Description

USART Driver library library should be configured via MHC. Below are the Snapshot of the MHC configuration window for USART driver and brief description.



USART Driver Configuration Asynchronous Mode



USART Driver Configuration Synchronous Mode

Common User Configuration for all Instances

1. Driver Mode

1. Allows User to select the mode of driver(**Asynchronous or Synchronous**). This setting is common for all the instances

Instance Specific User Configurations

1. PLIB Used

1. Indicates the underlying USART/UART PLIB used by the driver.

2. Number of clients

1. The total number of clients that can open the given USART driver instance.

2. Available only in Synchronous mode of Operations

3. Transmit Queue Size

1. Indicates the size of the transmit queue for the given USART/UART driver instance.
2. **Available only in Asynchronous mode of Operations**
4. **Receive Queue Size**
 1. Indicates the size of the receive queue for the given USART/UART driver instance.
 2. **Available only in Asynchronous mode of Operations**
5. **Use DMA for Transmit?**
 1. Enables DMA For transmitting the data
 2. **DMA Channel For Transmit**
 1. DMA Channel for transmission is automatically allocated in DMA configurations
6. **Use DMA for Receive?**
 1. Enables DMA For Receiving the data
 1. **DMA Channel For Receive**
 1. DMA Channel for Receiving is automatically allocated in DMA configurations

Building the Library

This section provides information on how the USART Driver Library can be built.

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Functions

	Name	Description
≡	DRV_USART_Initialize	Initializes the USART instance for the specified driver index.
≡	DRV_USART_Status	Gets the current status of the USART driver module.
≡	DRV_USART_SerialSetup	Sets the USART serial communication settings dynamically.

b) Core Client Functions

	Name	Description
≡	DRV_USART_Open	Opens the specified USART driver instance and returns a handle to it.
≡	DRV_USART_Close	Closes an opened-instance of the USART driver.

c) Data Transfer Functions

	Name	Description
≡	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.
≡	DRV_USART_BufferCompletedBytesGet	Returns the number of bytes that have been processed for the specified buffer request.
≡	DRV_USART_BufferStatusGet	Returns the transmit/receive request status.
≡	DRV_USART_ReadBuffer	This is a blocking function that reads data over USART.
≡	DRV_USART_ReadBufferAdd	Queues a read operation.
≡	DRV_USART_ReadQueuePurge	Removes all buffer requests from the read queue.
≡	DRV_USART_WriteBuffer	This is a blocking function that writes data over USART.
≡	DRV_USART_WriteBufferAdd	Queues a write operation.
≡	DRV_USART_WriteQueuePurge	Removes all buffer requests from the write queue.
≡	DRV_USART_ErrorGet	Gets the USART hardware errors associated with the client.

d) Data Types and Constants

	Name	Description
	DRV_USART_INIT	Defines the data required to initialize the USART driver
	DRV_USART_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the 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_ERROR	Defines the different types of errors for USART driver
	DRV_USART_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_USART_SERIAL_SETUP	Defines the data required to dynamically set the serial settings.

Description

This section describes the Application Programming Interface (API) 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.

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.

Remarks

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

This routine should only be called once during system initialization. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
// The following code snippet shows an example USART driver initialization.
SYS_MODULE_OBJ objectHandle;

const DRV_USART_PLIB_INTERFACE drvUsart0plibAPI = {
    .readCallbackRegister = (DRV_USART_PLIB_READ_CALLBACK_REG)USART1_ReadCallbackRegister,
    .read = (DRV_USART_PLIB_READ)USART1_Read,
    .readIsBusy = (DRV_USART_PLIB_READ_IS_BUSY)USART1_ReadIsBusy,
    .readCountGet = (DRV_USART_PLIB_READ_COUNT_GET)USART1_ReadCountGet,
    .writeCallbackRegister = (DRV_USART_PLIB_WRITE_CALLBACK_REG)USART1_WriteCallbackRegister,
```

```

    .write = (DRV_USART_PLIB_WRITE)USART1_Write,
    .writeIsBusy = (DRV_USART_PLIB_WRITE_IS_BUSY)USART1_WriteIsBusy,
    .writeCountGet = (DRV_USART_PLIB_WRITE_COUNT_GET)USART1_WriteCountGet,
    .errorGet = (DRV_USART_PLIB_ERROR_GET)USART1_ErrorGet,
    .serialSetup = (DRV_USART_PLIB_SERIAL_SETUP)USART1_SerialSetup
};

const DRV_USART_INIT drvUsart0InitData = {
    .usartPlib = &drvUsart0PlibAPI,
    .remapDataWidth = drvUsart0remapDataWidth,
    .remapParity = drvUsart0remapParity,
    .remapStopBits = drvUsart0remapStopBits,
    .remapError = drvUsart0remapError,

    .dmaChannelTransmit = SYS_DMA_CHANNEL_NONE,
    .dmaChannelReceive = SYS_DMA_CHANNEL_NONE,
    .numClients = DRV_USART_CLIENTS_NUMBER_IDX0,
    .clientObjPool = (uintptr_t)&drvUSART0ClientObjPool[0],
};

objectHandle = DRV_USART_Initialize(DRV_USART_INDEX_1,
    (SYS_MODULE_INIT*)&drvUsart0InitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}

```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to the init 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_Status Function

Gets the current status of the USART driver module.

File

[drv_usart.h](#)

C

```
SYS_STATUS DRV_USART_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Initialization have succeeded and the USART is ready for additional operations

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 be 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_SerialSetup Function

Sets the USART serial communication settings dynamically.

File

[drv_usart.h](#)

C

```
bool DRV_USART_SerialSetup(const DRV_HANDLE handle, DRV_USART_SERIAL_SETUP * setup);
```

Returns

true - Serial setup was updated successfully. false - Failure while updating serial setup.

Description

This function sets the USART serial communication settings dynamically.

Remarks

None.

Preconditions

[DRV_USART_Open](#) must have been called to obtain a valid opened device handle. The USART transmit or receive transfer status should not be busy.

Example

```
// 'handle', returned from the DRV_USART_Open

DRV_USART_SERIAL_SETUP setup = {
    115200,
    DRV_USART_DATA_8_BIT,
    DRV_USART_PARITY_ODD,
    DRV_USART_STOP_1_BIT
};

DRV_USART_SerialSetup(handle, &setup);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
setup	Pointer to the structure containing the serial setup.

Function

```
bool DRV_USART_SerialSetup(const DRV_HANDLE handle,
                           DRV_USART_SERIAL_SETUP * setup)
```

b) Core Client Functions

DRV_USART_Open Function

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

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 peripheral 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.

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 useof 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.

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. A new handle must be obtained by calling [DRV_USART_Open](#) before the caller may use the driver again.

Remarks

None.

Preconditions

[DRV_USART_Open](#) must have been called to obtain a valid opened device handle.

Example

```
// 'handle', returned from the DRV_USART_Open
DRV_USART_Close(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 )
```

c) Data Transfer Functions

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.

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 register a buffer event handling function with the driver to call back when queued buffer transfers have finished. When a client calls either the [DRV_USART_ReadBufferAdd](#) or [DRV_USART_WriteBufferAdd](#) 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

[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_USARTBufferEventHandler,
                                  (uintptr_t)&myAppObj );

DRV_USART_ReadBufferAdd(myUSARTHandle, myBuffer, MY_BUFFER_SIZE,
                       &bufferHandle);

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 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_COMPLETE:
        // This means the data was transferred.
        break;

    case DRV_USART_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_USART_BufferEventHandlerSet
(
    const DRV_HANDLE handle,
    const DRV_USART_BUFFER_EVENT_HANDLER eventHandler,
    const uintptr_t context
)

```

DRV_USART_BufferCompletedBytesGet Function

Returns the number of bytes that have been processed for the specified buffer request.

File

[drv_usart.h](#)

C

```
size_t DRV_USART_BufferCompletedBytesGet(DRV_USART_BUFFER_HANDLE bufferHandle);
```

Returns

Returns the number of bytes that have been processed for this buffer.

Returns [DRV_USART_BUFFER_HANDLE_INVALID](#) for an invalid or an expired buffer handle.

Description

The client can use this function, in a case where the buffer is terminated due to an error, to obtain the number of bytes that have been processed. Or in any other use case. This function can be used for non-DMA buffer transfers only. It cannot be used when the USART driver is configured to use DMA.

Remarks

This function is expected to work in non-DMA mode only. This function is thread safe when used in a RTOS application. If called from the callback, it must not call an OSAL mutex or critical section.

Preconditions

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

Either the `DRV_USART_ReadBufferAdd` or `DRV_USART_WriteBufferAdd` 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_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_USARTBufferEventHandler,
                                  (uintptr_t)&myAppObj );

DRV_USART_ReadBufferAdd( myUSARTHandle, myBuffer, MY_BUFFER_SIZE,
                        bufferHandle);

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_COMPLETE:
            // This means the data was transferred.
            break;

        case DRV_USART_BUFFER_EVENT_ERROR:
            // Error handling here.
            // We can find out how many bytes have been processed in this
            // buffer request prior to the error.
            processedBytes= DRV_USART_BufferCompletedBytesGet(bufferHandle);
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
bufferhandle	Handle for the buffer of which the processed number of bytes to be obtained.

Function

`size_t DRV_USART_BufferCompletedBytesGet`

```
(  
    DRV_USART_BUFFER_HANDLE bufferHandle  
)
```

DRV_USART_BufferStatusGet Function

Returns the transmit/receive request status.

File

[drv_usart.h](#)

C

```
DRV_USART_BUFFER_EVENT DRV_USART_BufferStatusGet(const DRV_USART_BUFFER_HANDLE bufferHandle);
```

Returns

Returns either pending, success or error event for the buffer. Pending means the buffer is queued but not serviced yet.

Description

This function can be used to poll the status of the queued buffer request if the application doesn't prefer to use the event handler (callback) function to get notified.

Remarks

This function returns error event if the buffer handle is invalid.

Preconditions

[DRV_USART_Open](#) must have been called to obtain a valid opened device handle.

Either the [DRV_USART_ReadBufferAdd](#) or [DRV_USART_WriteBufferAdd](#) 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_USART_BUFFER_HANDLE bufferHandle;  
DRV_USART_BUFFER_EVENT event;  
  
// 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_USARTBufferEventHandler,  
                                (uintptr_t)&myAppObj );  
  
DRV_USART_ReadBufferAdd( myUSARTHandle, myBuffer, MY_BUFFER_SIZE,  
                        bufferHandle );  
  
if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)  
{  
    // Error handling here  
}  
  
//Check the status of the buffer  
//This call can be used to wait until the buffer is processed.  
while ((event = DRV_USART_BufferStatusGet(bufferHandle)) == DRV_USART_BUFFER_EVENT_PENDING);  
//Buffer is processed, check the event variable to determine if the buffer request  
//is executed successfully or not.
```

Parameters

Parameters	Description
bufferhandle	Handle for the buffer of which the processed number of bytes to be obtained.

Function

```
DRV_USART_BUFFER_EVENT DRV_USART_BufferStatusGet
(
const DRV_USART_BUFFER_HANDLE bufferHandle
)
```

DRV_USART_ReadBuffer Function

This is a blocking function that reads data over USART.

File

[drv_usart.h](#)

C

```
bool DRV_USART_ReadBuffer(const DRV_HANDLE handle, void * buffer, const size_t size);
```

Returns

true - read is successful false - error has occurred

Description

This function does a blocking read operation. The function blocks till the data read is complete or error has occurred during read. Function will return false to report failure. The failure will occur for the following reasons:

- if the handle is invalid
- if the input buffer pointer is NULL
- if the buffer size is 0
- Hardware errors

Remarks

This function is thread safe in a RTOS application. This function should not be called from an interrupt context.

Preconditions

[DRV_USART_Open](#) must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t myBuffer[MY_BUFFER_SIZE];

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

if (DRV_USART_ReadBuffer(myUSARTHandle, myBuffer, MY_BUFFER_SIZE) == false)
{
    // Error handling here
}
```

Parameters

Parameters	Description
handle	Handle of the communication channel as return by the DRV_USART_Open function.
buffer	Pointer to the receive buffer.
size	Buffer size in bytes.

Function

```
bool DRV_USART_ReadBuffer
(
    const DRV_HANDLE handle,
    void * buffer,
    const size_t size
);
```

DRV_USART_ReadBufferAdd Function

Queues a read operation.

File

[drv_usart.h](#)

C

```
void DRV_USART_ReadBufferAdd(const DRV_HANDLE handle, void * buffer, const size_t size,
DRV_USART_BUFFER_HANDLE * const bufferHandle);
```

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 or [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

[DRV_USART_Open](#) must have been called to obtain a valid opened device handle.

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.

DRV_USART_ReadBufferAdd(myUSARTHandle, myBuffer, MY_BUFFER_SIZE,
&bufferHandle);
```

```

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

```

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.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

```

void DRV_USART_ReadBufferAdd
(
    const    DRV_HANDLE handle,
    void * buffer,
    const size_t size,
    DRV_USART_BUFFER_HANDLE * bufferHandle
)

```

DRV_USART_ReadQueuePurge Function

Removes all buffer requests from the read queue.

File

[drv_usart.h](#)

C

```
bool DRV_USART_ReadQueuePurge( const DRV_HANDLE handle );
```

Returns

True - Read queue purge is successful. False - Read queue purge has failed.

Description

This function removes all the buffer requests from the read queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

Preconditions

[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.
// Use DRV_USART_ReadBufferAdd to queue read requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_USART_ReadQueuePurge(myUSARTHandle))
    {
        //Couldn't purge the read queue, try again.

```

```

    }
    else
    {
        //Queue purge successful.
    }
}

```

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_USART_Open function.

Function

```
bool DRV_USART_ReadQueuePurge( const DRV\_HANDLE handle )
```

DRV_USART_WriteBuffer Function

This is a blocking function that writes data over USART.

File

[drv_usart.h](#)

C

```
bool DRV_USART_WriteBuffer(const DRV\_HANDLE handle, void * buffer, const size\_t size);
```

Returns

true - write is successful false - error has occurred

Description

This function does a blocking write operation. The function blocks till the data write is complete. Function will return false to report failure. The failure will occur for the following reasons:

- if the handle is invalid
- if the input buffer pointer is NULL
- if the buffer size is 0

Remarks

This function is thread safe in a RTOS application. This function should not be called from an interrupt context.

Preconditions

[DRV_USART_Open](#) must have been called to obtain a valid opened device handle.

Example

```

MY_APP_OBJ myAppObj;
uint8\_t myBuffer[MY\_BUFFER\_SIZE];

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

if (DRV_USART_WriteBuffer(myUSARTHandle, myBuffer, MY\_BUFFER\_SIZE) == false)
{
    // Error handling here
}

```

Parameters

Parameters	Description
handle	Handle of the communication channel as return by the DRV_USART_Open function.
buffer	Pointer to the data to be transmitted.

size	Buffer size in bytes.
------	-----------------------

Function

```
bool DRV_USART_WriteBuffer
(
const DRV_HANDLE handle,
void * buffer,
const size_t size
);
```

DRV_USART_WriteBufferAdd Function

Queues a write operation.

File

[drv_usart.h](#)

C

```
void DRV_USART_WriteBufferAdd(const DRV_HANDLE handle, void * buffer, const size_t size,
DRV_USART_BUFFER_HANDLE * bufferHandle);
```

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 driver 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

[DRV_USART_Open](#) must have been called to obtain a valid opened device handle.

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.

DRV_USART_WriteBufferAdd(myUSARTHandle, myBuffer, MY_BUFFER_SIZE,
&bufferHandle);
```

```

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

```

Parameters

Parameters	Description
handle	Handle of the communication channel as return by the DRV_USART_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_USART_WriteBufferAdd
(
    const DRV_HANDLE handle,
    void * buffer,
    size_t size,
    DRV_USART_BUFFER_HANDLE * bufferHandle
);

```

DRV_USART_WriteQueuePurge Function

Removes all buffer requests from the write queue.

File

[drv_usart.h](#)

C

```
bool DRV_USART_WriteQueuePurge(const DRV_HANDLE handle);
```

Returns

True - Write queue purge is successful. False - Write queue purge has failed.

Description

This function removes all the buffer requests from the write queue. The client can use this function to purge the queue on timeout or to remove unwanted stalled buffer requests or in any other use case.

Remarks

This function is thread safe when used in an RTOS environment. Avoid this function call from within the callback.

Preconditions

[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.
// Use DRV_USART_WriteBufferAdd to queue write requests

// Application timeout function, where remove queued buffers.
void APP_TimeOut(void)
{
    if(false == DRV_USART_WriteQueuePurge(myUSARTHandle))

```

```

    {
        //Couldn't purge the write queue, try again.
    }
    else
    {
        //Queue purge successful.
    }
}

```

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_USART_Open function.

Function

`bool DRV_USART_WriteQueuePurge(const DRV_HANDLE handle)`

DRV_USART_ErrorGet Function

Gets the USART hardware errors associated with the client.

File

[drv_usart.h](#)

C

```
DRV_USART_ERROR DRV_USART_ErrorGet( const DRV_HANDLE handle );
```

Returns

Errors occurred as listed by [DRV_USART_ERROR](#). This function reports multiple USART errors if occurred.

Description

This function returns the errors associated with the given client. The call to this function also clears all the associated error flags.

Remarks

USART errors are normally associated with the receiver. The driver clears all the errors internally and only returns the occurred error information for the client.

Preconditions

[DRV_USART_Open](#) must have been called to obtain a valid opened device handle.

Example

```

// 'handle', returned from the DRV_USART_Open

if (DRV_USART_ERROR_OVERRUN & DRV_USART_ErrorGet(handle))
{
    //Errors are cleared by the driver, take respective action
    //for the overrun error case.
}

```

Parameters

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

Function

`DRV_USART_ERROR DRV_USART_ErrorGet(const DRV_HANDLE handle)`

d) Data Types and Constants

DRV_USART_INIT Type

Defines the data required to initialize the USART driver

File

[drv_usart.h](#)

C

```
typedef struct _DRV_USART_INIT DRV_USART_INIT;
```

Description

USART Driver Initialization Data

This data type defines the data required to initialize the USART driver.

Remarks

This structure is implementation specific. It is fully defined in `drv_usart_definitions.h`.

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_ReadBufferAdd](#) or [DRV_USART_WriteBufferAdd](#) 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_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_PENDING,
DRV_USART_BUFFER_EVENT_COMPLETE,
DRV_USART_BUFFER_EVENT_ERROR
} DRV_USART_BUFFER_EVENT;

```

Members

Members	Description
DRV_USART_BUFFER_EVENT_PENDING	The buffer is pending to be serviced
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.

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_ReadBufferAdd](#) or [DRV_USART_WriteBufferAdd](#) 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 using the 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_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_BufferCompletedBytesGet](#) 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. And bufferHandle will be valid while the buffer request is in the queue and during callback, unless an error occurred. After callback returns, the driver will retire the buffer handle.

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 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_ReadBufferAdd](#) and [DRV_USART_WriteBufferAdd](#) 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,

USART2 driver buffers cannot be added 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 event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_USART_ERROR Type

Defines the different types of errors for USART driver

File

[drv_usart.h](#)

C

```
typedef enum _DRV_USART_ERROR DRV_USART_ERROR;
```

Description

USART Driver Errors

This data type defines the different types of errors for USART driver. DRV_USART_ERROR_NONE : No errors
 DRV_USART_ERROR_OVERRUN : Receive Overflow error
 DRV_USART_ERROR_PARITY : Parity error
 DRV_USART_ERROR_FRAMING : Framing error

Remarks

This structure is implementation specific. It is fully defined in `drv_usart_definitions.h`.

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_ReadBufferAdd](#) and [DRV_USART_WriteBufferAdd](#) functions if the buffer add request was not successful.

Remarks

None

DRV_USART_SERIAL_SETUP Type

Defines the data required to dynamically set the serial settings.

File

[drv_usart.h](#)

C

```
typedef struct _DRV_USART_SERIAL_SETUP DRV_USART_SERIAL_SETUP;
```

Description

USART Driver Serial Setup Data

This data type defines the data required to dynamically set the serial settings for the specific USART driver instance.

Remarks

This structure is implementation specific. It is fully defined in [drv_usart_definitions.h](#).

Files**Files**

Name	Description
drv_usart.h	USART Driver Interface Header File

Description

This section will list only the library's interface header file(s).

drv_usart.h

USART Driver Interface Header File

Enumerations

	Name	Description
	DRV_USART_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.

Functions

	Name	Description
≡◊	DRV_USART_BufferCompletedBytesGet	Returns the number of bytes that have been processed for the specified buffer request.
≡◊	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.

DRV_USART_BufferStatusGet	Returns the transmit/receive request status.
DRV_USART_Close	Closes an opened-instance of the USART driver.
DRV_USART_ErrorGet	Gets the USART hardware errors associated with the client.
DRV_USART_Initialize	Initializes the USART instance for the specified driver index.
DRV_USART_Open	Opens the specified USART driver instance and returns a handle to it.
DRV_USART_ReadBuffer	This is a blocking function that reads data over USART.
DRV_USART_ReadBufferAdd	Queues a read operation.
DRV_USART_ReadQueuePurge	Removes all buffer requests from the read queue.
DRV_USART_SerialSetup	Sets the USART serial communication settings dynamically.
DRV_USART_Status	Gets the current status of the USART driver module.
DRV_USART_WriteBuffer	This is a blocking function that writes data over USART.
DRV_USART_WriteBufferAdd	Queues a write operation.
DRV_USART_WriteQueuePurge	Removes all buffer requests from the write queue.

Macros

	Name	Description
	DRV_USART_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.

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_ERROR	Defines the different types of errors for USART driver
	DRV_USART_INIT	Defines the data required to initialize the USART driver
	DRV_USART_SERIAL_SETUP	Defines the data required to dynamically set the serial settings.

Description

USART Driver Interface Header File

The USART device driver provides a simple interface to manage the USART or UART modules on Microchip PIC32 microcontrollers. This file provides the interface definition for the USART driver.

File Name

drv_usart.h

Company

Microchip Technology Inc.

System Service Libraries Help

This section provides help for the System Service libraries that are available in MPLAB Harmony 3 Core.

Command Processor System Service Library

This section describes the Command Processor System Service Library.

Introduction

This library provides an abstraction of the Command Processor System Service Library with a convenient C language interface. It provides the framework for a command console that can support commands from multiple client sources.

Description

The Command Processor System Service provides the developer with simple APIs to implement a command console. The console may support commands from one or more client software modules.

Using the Library

This topic describes the basic architecture of the Command Processor System Service Library and provides information on its use.

Description

In conjunction with the Console System Service, the Command Processor System Service provides the user with an ASCII command prompt. It will interpret the commands entered at the prompt and process accordingly. It also supports command history, as well as command help.

Abstraction Model

This topic provides a description of the software abstraction for the Command Processor System Service.

Description

The Command Processor System Service Library is a collection of operations specific to supporting user input commands. The commands can be uniquely native to each client service and can be dynamically added. The library can support multiple client services at once.

The Command Processor System Service is a module that works closely with the Console System Service to present a user interface command prompt.

How the Library Works

This topic describes the basic architecture of the Command System Service Library and provides information on its implementation.

Description

Initialization and Reinitialization

Initialization of the Command Processor System Service initializes the status of the module and sets the state of the internal state machine.

The Command Processor Status and Tasks routines are required for the normal operation of the service.

Adding Commands

The Command Processor System Service will accept commands dynamically during run-time through the follow interface:

```
SYS_COMMAND_ADDGRP(const _SYS_CMD_DCPT* pCmdTbl, int nCmds, const char*  
groupName, const char* menuStr)
```

Command Prompt

In conjunction with the Console System Service, the Command Processor System Service provides the user with an ASCII command prompt. It will interpret the commands entered at the prompt and process accordingly. It also supports command history, as well as command help.

Configuring the Library

This section provides information on how to configure the Command System Service library.

Description

The Command Processor System Service library should be configured via MHC. Below is the snapshot of the MHC configuration window and a brief description of various configuration options.

In MHC, the Command Processor System Service configuration options are available under the Console System Service component.

Configuration Options:

- **Device Used** - Indicates the hardware UART Peripheral Library instance used by the Console System Service
- **Transmit Buffer Queue Size (1-128)** - Indicates the maximum number of data transmit requests that can be queued
- **Receive Buffer Queue Size (1-128)** - Indicates the maximum number of data reception requests that can be queued
- **Enable Debug?** - Enable Debug System Service. Refer the Debug System Service for the Debug service configuration options.
- **Enable Command Processor?** - Enable Command Processor System Service
 - **Command Print Buffer Size (512-8192)** - Size of the command print buffer in bytes
 - **Re-route Console Message/Print through Command Service?** - Re-routes the **SYS_MESSAGE**, **SYS_CONSOLE_MESSAGE**, **SYS_PRINT** and **SYS_CONSOLE_PRINT** macros through the Command Service.
 - **Re-route Debug Message/Print through Command Service?** - Re-routes the **SYS_DEBUG_MESSAGE**, **SYS_DEBUG_PRINT**, **SYS_ERROR_PRINT**, **SYS_DEBUG** and **SYS_ERROR** macros through the Command Service. For re-routing debug messages through command service, the debug system service must be enabled in MHC configuration.

Building the Library

This section provides information on how the Command Processor System Service Library can be built.

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Interaction Functions

	Name	Description
≡	SYS_CMD_ADDGRP	Allows clients to add command process to the Command Processor System Service.
≡	SYS_CMD_DELETE	Removes CMDIO parameters from the Command Processor System Service console.
≡	SYS_CMD_MESSAGE	Outputs a message to the Command Processor System Service console.
≡	SYS_CMD_PRINT	Outputs a printout to the Command Processor System Service console.
≡	SYS_CMD_READY_TO_READ	Allows upper layer application to confirm that the command module is ready to accept command input

	SYS_CMD_READY_TO_WRITE	Allows upper layer application to confirm that the command module is ready to write output to the Console System Service.
	SYS_CMD_Tasks	Maintains the Command Processor System Service's internal state machine.
	SYS_CMD_Initialize	Initializes data for the instance of the Command Processor module.
	SYS_CMD_RegisterCallback	Registers a callback function with the command service that will be executed when the lower layer read or write queue is empty.

b) Command I/O Functions

	Name	Description
	SYS_CMDIO_GET_HANDLE	Gets the CMDIO Handle base via index.
	SYS_CMDIO_ADD	Adds CMDIO parameters to the Command Processor System Service console.

c) Data Types and Constants

	Name	Description
	_promptStr	prompt string
	COMMAND_HISTORY_DEPTH	Command Processor System Service Maximum Depth of Command History.
	ESC_SEQ_SIZE	standard VT100 escape sequences
	LINE_TERM	line terminator
	MAX_CMD_ARGS	Command Processor System Service Maximum Number of Argument definitions.
	MAX_CMD_GROUP	Command Processor System Service Maximum Number of Command Group definitions.
	SYS_CMD_MAX_LENGTH	Command Processor System Service Command Buffer Maximum Length definition.
	SYS_CMD_READ_BUFFER_SIZE	Command Processor System Service Read Buffer Size definition.
	SYS_CMD_DEVICE_NODE	Defines the data structure to store each command instance.
	SYS_CMD_API	Identifies the Command API structure.
	SYS_CMD_DATA_RDY_FNC	Identifies a data available function API.
	SYS_CMD_DESCRIPTOR	a simple command descriptor
	SYS_CMD_DESCRIPTOR_TABLE	table containing the supported commands
	SYS_CMD_DEVICE_LIST	Defines the list structure to store a list of command instances.
	SYS_CMD_FNC	Identifies the command process function API.
	SYS_CMD_GETC_FNC	Identifies a get single character function API.
	SYS_CMD_HANDLE	Identifies a particular Command I/O instance.
	SYS_CMD_INIT_DATA	Defines the data structure to store each command.
	SYS_CMD_MSG_FNC	Identifies a message function API.
	SYS_CMD_PRINT_FNC	Identifies a print function API.
	SYS_CMD_PUTC_FNC	Identifies a single character print function API.
	SYS_CMD_READC_FNC	Identifies a read single character function API.
	SYS_CMD_STATE	Defines the various states that can be achieved by a Command instance.
	SYS_CMD_BUFFER_DMA_READY	Define this for MX #define SYS_CMD_BUFFER_DMA_READY __attribute__((coherent)) __attribute__((aligned(4))) //Define this for MZ
	SYS_DEBUG_MESSAGE	Prints a debug message if the system error level is defined at or lower than the level specified.
	SYS_DEBUG_PRINT	Formats and prints an error message if the system error level is defined at or lower than the level specified.
	SYS_ERROR_PRINT	This is macro SYS_ERROR_PRINT.
	SYS_MESSAGE	Prints a message to the console regardless of the system error level.
	SYS_PRINT	Formats and prints an error message with a variable number of arguments regardless of the system error level.
	SYS_CMD_CallbackFunction	Command Callback Function Handle.

	SYS_CMD_CONSOLE_IO_PARAM	Defines whether the command module is single character or full command read
	SYS_CMD_EVENT	Identifies the Command Event Types
	SYS_CMD_INIT	Identifies the system command initialize structure.
	SYS_CMD_DEVICE_MAX_INSTANCES	This is macro SYS_CMD_DEVICE_MAX_INSTANCES.
	SYS_CMD_MESSAGE	This is macro SYS_CMD_MESSAGE.
	SYS_CMD_PRINT	This is macro SYS_CMD_PRINT.
	SYS_DEBUG	
	SYS_ERROR	This is macro SYS_ERROR.

Description

This section describes the APIs of the Command Processor System Service Library.

Refer to each section for a detailed description.

a) System Interaction Functions

SYS_CMD_ADDGRP Function

Allows clients to add command process to the Command Processor System Service.

File

[sys_command.h](#)

C

```
bool SYS_CMD_ADDGRP(const SYS_CMD_DESCRIPTOR* pCmdTbl, int nCmds, const char* groupName, const char* menuStr);
```

Returns

- true - Indicates success
- false - Indicates an error occurred

Description

Client modules may call this function to add command process to the Command Processor System Service.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Example

```
static const SYS_CMD_DESCRIPTOR cmdTbl[] = { {"command_as_typed_at_the_prompt",  
_Function_Name_That_Supports_The_Command, ": Helpful description of the command for the user"}, };  
bool APP_AddCommandFunction() { if (!SYS_CMD_ADDGRP(cmdTbl, sizeof(cmdTbl)/sizeof(*cmdTbl), "Command Group  
Name", ": Command Group Description")) { return false; } return true; }
```

Function

```
bool SYS_CMD_ADDGRP(const SYS_CMD_DESCRIPTOR* pCmdTbl, int nCmds,  
const char* groupName,  
const char* menuStr)
```

SYS_CMD_DELETE Function

Removes CMDIO parameters from the Command Processor System Service console.

File

[sys_command.h](#)

C

```
bool SYS_CMD_DELETE(SYS_CMD_DEVICE_NODE* pDevNode);
```

Returns

None.

Description

This function removes CMDIO parameters from the Command Processor System Service console.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
bool SYS_CMD_DELETE( SYS_CMD_DEVICE_NODE* pDevNode);
```

SYS_CMD_MESSAGE Function

Outputs a message to the Command Processor System Service console.

File

[sys_command.h](#)

C

```
void SYS_CMD_MESSAGE(const char* message);
```

Returns

None.

Description

This function outputs a message to the Command Processor System Service console.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
void SYS_CMD_MESSAGE (const char* message)
```

SYS_CMD_PRINT Function

Outputs a printout to the Command Processor System Service console.

File[sys_command.h](#)**C**

```
void SYS_CMD_PRINT(const char * format, ...);
```

Returns

None.

Description

This function outputs a printout to the Command Processor System Service console.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
void SYS_CMD_PRINT(const char *format, ...)
```

SYS_CMD_READY_TO_READ Function

Allows upper layer application to confirm that the command module is ready to accept command input

File[sys_command.h](#)**C**

```
bool SYS_CMD_READY_TO_READ();
```

Returns

- true - Indicates command module is ready
- false - Indicates command module is not ready

Description

This function allows upper layer application to confirm that the command module is ready to accept command input

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
bool SYS_CMD_READY_TO_READ( void )
```

SYS_CMD_READY_TO_WRITE Function

Allows upper layer application to confirm that the command module is ready to write output to the Console System Service.

File[sys_command.h](#)

C

```
bool SYS_CMD_READY_TO_WRITE();
```

Returns

- true - Indicates command module is ready
- false - Indicates command module is not ready

Description

This function allows upper layer application to confirm that the command module is ready to write output to the Console System Service.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
bool SYS_CMD_READY_TO_WRITE( void )
```

SYS_CMD_Tasks Function

Maintains the Command Processor System Service's internal state machine.

File

[sys_command.h](#)

C

```
bool SYS_CMD_Tasks();
```

Returns

- true - Indicates success
- false - Indicates an error occurred

Description

This function is used to maintain the Command Processor System Service internal state machine.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
bool SYS_CMD_Tasks( void )
```

SYS_CMD_Initialize Function

Initializes data for the instance of the Command Processor module.

File

[sys_command.h](#)

C

```
bool SYS_CMD_Initialize(const SYS_MODULE_INIT * const init);
```

Returns

- true - Indicates success
- false - Indicates an error occurred

Description

This function initializes the Command Processor module. It also initializes any internal data structures.

Remarks

This routine should only be called once during system initialization.

Preconditions

None.

Parameters

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

Function

```
bool SYS_CMD_Initialize( const SYS\_MODULE\_INIT * const init )
```

SYS_CMD_RegisterCallback Function

Registers a callback function with the command service that will be executed when the lower layer read or write queue is empty.

File

[sys_command.h](#)

C

```
void SYS_CMD_RegisterCallback(SYS\_CMD\_CallbackFunction cbFunc, SYS\_CMD\_EVENT event);
```

Returns

None.

Description

This function is used by an application to register a callback function with the command service. The callback function is called in response to an event. Separate callback functions are required for each event.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
cbFunc	The name of the callback function
event	Enumerated list of events that can trigger a callback

Function

```
void SYS_CMD_RegisterCallback(SYS\_CMD\_CallbackFunction cbFunc, SYS\_CMD\_EVENT event)
```

b) Command I/O Functions

SYS_CMDIO_GET_HANDLE Function

Gets the CMDIO Handle base via index.

File

[sys_command.h](#)

C

```
SYS_CMD_DEVICE_NODE* SYS_CMDIO_GET_HANDLE(short num);
```

Returns

- [SYS_CMD_DEVICE_NODE](#) Handle for the CMDIO - Indicates success
- NULL - Indicates not successful

Description

This function returns the handle for the CMDIO when supplied with an index.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
SYS_CMD_DEVICE_NODE* SYS_CMDIO_GET_HANDLE(short num)
```

SYS_CMDIO_ADD Function

Adds CMDIO parameters to the Command Processor System Service console.

File

[sys_command.h](#)

C

```
SYS_CMD_DEVICE_NODE* SYS_CMDIO_ADD(const SYS_CMD_API* opApi, const void* cmdIoParam, const
SYS_CMD_CONSOLE_IO_PARAM cmdIoType);
```

Returns

None.

Description

This function adds CMDIO Parameters to the Command Processor System Service console.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
SYS_CMD_DEVICE_NODE* SYS_CMDIO_ADD(const SYS_CMD_API* opApi, const void* cmdIoParam,
const SYS_CMD_CONSOLE_IO_PARAM cmdIoType)
```

c) Data Types and Constants

_promptStr Macro

File

[sys_command.h](#)

C

```
#define _promptStr ">"           // prompt string
```

Description

prompt string

COMMAND_HISTORY_DEPTH Macro

Command Processor System Service Maximum Depth of Command History.

File

[sys_command.h](#)

C

```
#define COMMAND_HISTORY_DEPTH 3
```

Description

SYS CMD Processor Command History Depth

This macro defines the maximum depth of the command history.

Remarks

None.

ESC_SEQ_SIZE Macro

File

[sys_command.h](#)

C

```
#define ESC_SEQ_SIZE 2           // standard VT100 escape sequences
```

Description

standard VT100 escape sequences

LINE_TERM Macro

File

[sys_command.h](#)

C

```
#define LINE_TERM "\r\n"         // line terminator
```

Description

line terminator

MAX_CMD_ARGS Macro

Command Processor System Service Maximum Number of Argument definitions.

File

[sys_command.h](#)

C

```
#define MAX_CMD_ARGS 15
```

Description

SYS CMD Processor Maximum Number of Command Arguments

This macro defines the maximum number of arguments per command.

Remarks

None.

MAX_CMD_GROUP Macro

Command Processor System Service Maximum Number of Command Group definitions.

File

[sys_command.h](#)

C

```
#define MAX_CMD_GROUP 8
```

Description

SYS CMD Processor Maximum Number of Command Group

This macro defines the maximum number of command groups.

Remarks

None.

SYS_CMD_MAX_LENGTH Macro

Command Processor System Service Command Buffer Maximum Length definition.

File

[sys_command.h](#)

C

```
#define SYS_CMD_MAX_LENGTH 128
```

Description

SYS CMD Processor Buffer Maximum Length

This macro defines the maximum length of the command buffer.

Remarks

None.

SYS_CMD_READ_BUFFER_SIZE Macro

Command Processor System Service Read Buffer Size definition.

File

[sys_command.h](#)

C

```
#define SYS_CMD_READ_BUFFER_SIZE 128
```

Description

SYS CMD Processor Read Buffer Size

This macro defines the maximum size of the command buffer.

Remarks

None.

SYS_CMD_DEVICE_NODE Structure

Defines the data structure to store each command instance.

File

[sys_command.h](#)

C

```
struct SYS_CMD_DEVICE_NODE {
    char* cmdPnt;
    char* cmdEnd;
    char cmdBuff SYS_CMD_BUFFER_DMA_READY[SYS_CMD_MAX_LENGTH+1];
    const SYS_CMD_API* pCmdApi;
    const void* cmdIoParam;
    SYS_CMD_CONSOLE_IO_PARAM cmdIoType;
    struct SYS_CMD_DEVICE_NODE* next;
    SYS_CMD_STATE cmdState;
};
```

Members

Members	Description
const SYS_CMD_API* pCmdApi;	Cmd IO APIs
const void* cmdIoParam;	channel specific parameter

Description

SYS CMD Command Instance Node Structure

This data structure stores all the data relevant to a uniquely entered command instance. It is a node for a linked list structure to support the Command Processor System Service's command history feature

Remarks

None.

SYS_CMD_API Structure

Identifies the Command API structure.

File[sys_command.h](#)**C**

```
typedef struct {
    SYS_CMD_MSG_FNC msg;
    SYS_CMD_PRINT_FNC print;
    SYS_CMD_PUTC_FNC putc;
    SYS_CMD_DATA_RDY_FNC isRdy;
    SYS_CMD_GETC_FNC getc;
    SYS_CMD_READC_FNC readc;
} SYS_CMD_API;
```

Members

Members	Description
SYS_CMD_MSG_FNC msg;	Message function API
SYS_CMD_PRINT_FNC print;	Print function API
SYS_CMD_PUTC_FNC putc;	Put single char function API
SYS_CMD_DATA_RDY_FNC isRdy;	Data available API
SYS_CMD_GETC_FNC getc;	Get single data API
SYS_CMD_READC_FNC readc;	Read single data API

Description

SYS CMD API structure

This structure identifies the Command API structure.

Remarks

None.

SYS_CMD_DATA_RDY_FNC Type

Identifies a data available function API.

File[sys_command.h](#)**C**

```
typedef bool ( * SYS_CMD_DATA_RDY_FNC )(const void* cmdIoParam);
```

Description

Ready Status Check function API. This handle identifies the interface structure of the data available function API within the Command IO encapsulation.

Remarks

None.

SYS_CMD_DESCRIPTOR Structure**File**[sys_command.h](#)**C**

```
typedef struct {
    const char* cmdStr;
```

```

SYS_CMD_FNC cmdFnc;
const char* cmdDescr;
} SYS_CMD_DESCRIPTOR;

```

Members

Members	Description
const char* cmdStr;	string identifying the command
SYS_CMD_FNC cmdFnc;	function to execute for this command
const char* cmdDescr;	simple command description

Description

a simple command descriptor

SYS_CMD_DESCRIPTOR_TABLE Structure

File

[sys_command.h](#)

C

```

typedef struct {
    int nCmds;
    const SYS_CMD_DESCRIPTOR* pCmd;
    const char* cmdGroupName;
    const char* cmdMenuStr;
} SYS_CMD_DESCRIPTOR_TABLE;

```

Members

Members	Description
int nCmds;	number of commands available in the table
const SYS_CMD_DESCRIPTOR* pCmd;	pointer to an array of command descriptors
const char* cmdGroupName;	name identifying the commands
const char* cmdMenuStr;	help string

Description

table containing the supported commands

SYS_CMD_DEVICE_LIST Structure

Defines the list structure to store a list of command instances.

File

[sys_command.h](#)

C

```

typedef struct {
    int num;
    SYS_CMD_DEVICE_NODE* head;
    SYS_CMD_DEVICE_NODE* tail;
} SYS_CMD_DEVICE_LIST;

```

Description

SYS CMD Command List Structure

This data structure defines the linked list structure to support the Command Processor System Service's command history feature

Remarks

None.

SYS_CMD_FNC Type

Identifies the command process function API.

File

[sys_command.h](#)

C

```
typedef int (* SYS_CMD_FNC)(SYS_CMD_DEVICE_NODE* pCmdIO, int argc, char **argv);
```

Description

SYS CMD Command Function

Command Process Function API. This handle identifies the interface structure of the command process function API.

Remarks

None.

SYS_CMD_GETC_FNC Type

Identifies a get single character function API.

File

[sys_command.h](#)

C

```
typedef char (* SYS_CMD_GETC_FNC)(const void* cmdIoParam);
```

Description

Get Single Character function API. This handle identifies the interface structure of the get single character function API within the Command I/O encapsulation.

Remarks

None.

SYS_CMD_HANDLE Type

Identifies a particular Command I/O instance.

File

[sys_command.h](#)

C

```
typedef const void* SYS_CMD_HANDLE;
```

Description

Command I/O Handle. This event handle identifies a registered instance of a Command IO object. Every time the application that tries to access the parameters with respect to a particular event, this event handle is used to refer to that event.

Remarks

None.

SYS_CMD_INIT_DATA Structure

Defines the data structure to store each command.

File

[sys_command.h](#)

C

```
typedef struct {
    size_t bytesRead;
    int seqBytesRead;
    char seqBuff[ESC_SEQ_SIZE + 1];
    SYS_MODULE_INDEX moduleIndices[SYS_CMD_DEVICE_MAX_INSTANCES];
    int moduleInFd;
    int moduleOutFd;
} SYS_CMD_INIT_DATA;
```

Description

SYS CMD Command App Init Structure

This data structure stores all the data relevant to a uniquely entered command. It is a node for a linked list structure to support the command history functionality

Remarks

None.

SYS_CMD_MSG_FNC Type

Identifies a message function API.

File

[sys_command.h](#)

C

```
typedef void (* SYS_CMD_MSG_FNC)(const void* cmdIoParam, const char* str);
```

Description

Message function API. This handle identifies the interface structure of the message function API within the Command I/O encapsulation.

Remarks

None.

SYS_CMD_PRINT_FNC Type

Identifies a print function API.

File

[sys_command.h](#)

C

```
typedef void (* SYS_CMD_PRINT_FNC)(const void* cmdIoParam, const char* format, ...);
```

Description

Print function API. This handle identifies the interface structure of the print function API within the Command I/O encapsulation.

Remarks

None.

SYS_CMD_PUTC_FNC Type

Identifies a single character print function API.

File

[sys_command.h](#)

C

```
typedef void (* SYS_CMD_PUTC_FNC)(const void* cmdIoParam, char c);
```

Description

Single Character Print function API. This handle identifies the interface structure of single character print function API within the Command I/O encapsulation.

Remarks

None.

SYS_CMD_READC_FNC Type

Identifies a read single character function API.

File

[sys_command.h](#)

C

```
typedef size_t (* SYS_CMD_READC_FNC)(const void* cmdIoParam);
```

Description

Read Single Character function API

This handle identifies the interface structure of read single character function API within the Command I/O encapsulation.

Remarks

None.

SYS_CMD_STATE Enumeration

Defines the various states that can be achieved by a Command instance.

File

[sys_command.h](#)

C

```
typedef enum {
    SYS_CMD_STATE_DISABLE,
    SYS_CMD_STATE_SETUP_READ,
    SYS_CMD_STATE_WAIT_FOR_READ_DONE,
    SYS_CMD_STATE_PROCESS_FULL_READ
} SYS_CMD_STATE;
```

Description

SYS CMD State Machine States

This enumeration defines the various states that can be achieved by the command operation.

Remarks

None.

SYS_CMD_BUFFER_DMA_READY Macro

File

[sys_command.h](#)

C

```
#define SYS_CMD_BUFFER_DMA_READY
```

Description

Define this for MX #define SYS_CMD_BUFFER_DMA_READY __attribute__((coherent)) __attribute__((aligned(4))) //Define this for MZ

SYS_DEBUG_MESSAGE Macro

Prints a debug message if the system error level is defined at or lower than the level specified.

File

[sys_debug.h](#)

C

```
#define SYS_DEBUG_MESSAGE(level,message)
```

Returns

None.

Description

This function prints a debug message if the system error level is defined at or lower than the level specified. If mapped to the [SYS_DEBUG_Message](#) function, then the system debug service must be initialized and running.

Remarks

By default, this macro is defined as nothing, effectively removing all code generated by calls to it. To process SYS_DEBUG_MESSAGE calls, this macro must be defined in a way that maps calls to it to the desired implementation (see example, above).

This macro can be mapped to the system console service (along with other system debug macros) by defining SYS_DEBUG_USE_CONSOLE in the system configuration (configuration.h) instead of defining it individually.

Preconditions

If mapped to the [SYS_DEBUG_Message](#) function, then the system debug service must be initialized and running.

Example

```
// In configuration.h file: #define SYS_DEBUG_USE_CONSOLE
// In sys_debug.h file: #define SYS_DEBUG_MESSAGE(level, message) _SYS_DEBUG_MESSAGE(level, message)

SYS_DEBUG_ErrorLevelSet(SYS_ERROR_DEBUG);
SYS_DEBUG_MESSAGE(SYS_ERROR_WARNING, "System Debug Message rn");
```

Parameters

Parameters	Description
level	The current error level threshold for displaying the message.
message	Pointer to a buffer containing the message to be displayed.

Function

`SYS_DEBUG_MESSAGE(SYS_ERROR_LEVEL level, const char* message)`

SYS_DEBUG_PRINT Macro

Formats and prints an error message if the system error level is defined at or lower than the level specified.

File

`sys_debug.h`

C

`#define SYS_DEBUG_PRINT(level, fmt, ...)`

Returns

None.

Description

Macro: `SYS_DEBUG_PRINT(SYS_ERROR_LEVEL level, const char* format, ...)`

This macro formats and prints an error message if the system error level is defined at or lower than the level specified.

Remarks

The format string and arguments follow the printf convention.

By default, this macro is defined as nothing, effectively removing all code generated by calls to it. To process `SYS_DEBUG_PRINT` calls, this macro must be defined in a way that maps calls to it to the desired implementation (see example, above).

This macro can be mapped to the system console service (along with other system debug macros) by defining `SYS_DEBUG_USE_CONSOLE` in the system configuration (configuration.h) instead of defining it individually.

Preconditions

If mapped to the `SYS_DEBUG_Print` function, then the system debug service must be initialized and running.

Example

```
// In configuration.h file: #define SYS_DEBUG_USE_CONSOLE
// In sys_debug.h: #define SYS_DEBUG_PRINT(level, fmt, ...) _SYS_DEBUG_PRINT(level, fmt,
##__VA_ARGS__)

// In source code
int result;

result = SomeOperation();
if (result > MAX_VALUE)
{
    SYS_DEBUG_PRINT(SYS_ERROR_WARNING, "Result of %d exceeds max value", result);
    // Take appropriate action
}
```

Parameters

Parameters	Description
level	The current error level threshold for displaying the message.
format	Pointer to a buffer containing the format string for the message to be displayed.
...	Zero or more optional parameters to be formatted as defined by the format string.

SYS_ERROR_PRINT Macro

File

[sys_command.h](#)

C

```
#define SYS_ERROR_PRINT(level, fmt, ...) do { if((level) <= gblErrLvl) SYS_CMD_PRINT(fmt, ##__VA_ARGS__); } while (0)
```

Description

This is macro SYS_ERROR_PRINT.

SYS_MESSAGE Macro

Prints a message to the console regardless of the system error level.

File

[sys_debug.h](#)

C

```
#define SYS_MESSAGE(message)
```

Returns

None.

Description

Macro: SYS_MESSAGE(const char* message)

This macro is used to print a message to the console regardless of the system error level. It can be mapped to any desired implementation.

Remarks

By default, this macro is defined as nothing, effectively removing all code generated by calls to it. To process SYS_MESSAGE calls, this macro must be defined in a way that maps calls to it to the desired implementation (see example, above).

This macro can be mapped to the system console service (along with other system debug macros) by defining SYS_DEBUG_USE_CONSOLE in the system configuration (configuration.h) instead of defining it individually.

Preconditions

If mapped to the [SYS_DEBUG_Message](#) function, then the system debug service must be initialized and running.

Example

```
// In configuration.h file: #define SYS_DEBUG_USE_CONSOLE
// In sys_debug.h file: #define SYS_MESSAGE(message) SYS_DEBUG_Message(message)

// In source code
SYS_MESSAGE("My Message\n");
```

Parameters

Parameters	Description
message	Pointer to a buffer containing the message string to be displayed.

SYS_PRINT Macro

Formats and prints an error message with a variable number of arguments regardless of the system error level.

File[sys_debug.h](#)**C**

```
#define SYS_PRINT(fmt, ...)
```

Returns

None.

Description

This function formats and prints an error message with a variable number of arguments regardless of the system error level.

Remarks

The format string and arguments follow the printf convention.

By default, this macro is defined as nothing, effectively removing all code generated by calls to it. To process SYS_PRINT calls, this macro must be defined in a way that maps calls to it to the desired implementation (see example, above).

This macro can be mapped to the system console service (along with other system debug macros) by defining SYS_DEBUG_USE_CONSOLE in the system configuration (configuration.h) instead of defining it individually.

Preconditions

If mapped to the [SYS_DEBUG_Print](#) function, then the system debug service must be initialized and running.

Example

```
// In configuration.h file: #define SYS_DEBUG_USE_CONSOLE
// In sys_debug.h file: #define SYS_PRINT(fmt, ...) SYS_DEBUG_Print(fmt, ##__VA_ARGS__)

// In source code
int result;

result = SomeOperation();
if (result > MAX_VALUE)
{
    SYS_PRINT("Result of %d exceeds max value", result);
    // Take appropriate action
}
```

Parameters

Parameters	Description
format	Pointer to a buffer containing the format string for the message to be displayed.
...	Zero or more optional parameters to be formatted as defined by the format string.

Function[SYS_PRINT\(const char* format, ...\)](#)**SYS_CMD_CallbackFunction Type**

Command Callback Function Handle.

File[sys_command.h](#)**C**

```
typedef void (* SYS_CMD_CallbackFunction)(void *handle);
```

Description

Command Callback Function Handle.

Remarks

None.

SYS_CMD_CONSOLE_IO_PARAM Enumeration

Defines whether the command module is single character or full command read

File

[sys_command.h](#)

C

```
typedef enum {
    SYS_CMD_SINGLE_CHARACTER_READ_CONSOLE_IO_PARAM = 0,
    SYS_CMD_FULL_COMMAND_READ_CONSOLE_IO_PARAM = 1,
    SYS_CMD_TELNET_COMMAND_READ_CONSOLE_IO_PARAM = 2
} SYS_CMD_CONSOLE_IO_PARAM;
```

Description

SYS CMD Console I/O Parameter

This enumeration defines whether the command module is single character or full command read.

Remarks

None.

SYS_CMD_EVENT Enumeration

Identifies the Command Event Types

File

[sys_command.h](#)

C

```
typedef enum {
    SYS_CMD_EVENT_WRITE_COMPLETE,
    SYS_CMD_EVENT_READ_COMPLETE
} SYS_CMD_EVENT;
```

Description

SYS CMD EVENT structure

This structure identifies the Command Event Types.

Remarks

None.

SYS_CMD_INIT Structure

Identifies the system command initialize structure.

File

[sys_command.h](#)

C

```
typedef struct {
    SYS_MODULE_INIT moduleInit;
```

```

    uint8_t consoleCmdIOParam;
    SYS_CMD_CONSOLE_IO_PARAM cmdIoType;
} SYS_CMD_INIT;

```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization

Description

SYS CMD INIT structure

This structure identifies the system command initialize structure.

Remarks

None.

SYS_CMD_DEVICE_MAX_INSTANCES Macro

File

[sys_command.h](#)

C

```
#define SYS_CMD_DEVICE_MAX_INSTANCES 1
```

Description

This is macro SYS_CMD_DEVICE_MAX_INSTANCES.

SYS_CMD_MESSAGE Macro

File

[sys_command.h](#)

C

```
#define SYS_CMD_MESSAGE(message)
```

Description

This is macro SYS_CMD_MESSAGE.

SYS_CMD_PRINT Macro

File

[sys_command.h](#)

C

```
#define SYS_CMD_PRINT(fmt, ...)
```

Description

This is macro SYS_CMD_PRINT.

SYS_DEBUG Macro**File**[sys_command.h](#)**C**

```
#define SYS_DEBUG(level,message) SYS_DEBUG_MESSAGE(level,message)
```

Section

Data Types

SYS_ERROR Macro**File**[sys_command.h](#)**C**

```
#define SYS_ERROR(level,fmt, ...) SYS_ERROR_PRINT(level, fmt, ##__VA_ARGS__)
```

Description

This is macro SYS_ERROR.

Files**Files**

Name	Description
sys_command.h	Command Processor System Service interface definition.

Description

This section lists the header files used by the library.

sys_command.h

Command Processor System Service interface definition.

Enumerations

	Name	Description
	SYS_CMD_CONSOLE_IO_PARAM	Defines whether the command module is single character or full command read
	SYS_CMD_EVENT	Identifies the Command Event Types
	SYS_CMD_STATE	Defines the various states that can be achieved by a Command instance.

Functions

	Name	Description
≡	SYS_CMD_ADDGRP	Allows clients to add command process to the Command Processor System Service.
≡	SYS_CMD_DELETE	Removes CMDIO parameters from the Command Processor System Service console.
≡	SYS_CMD_Initialize	Initializes data for the instance of the Command Processor module.
≡	SYS_CMD_MESSAGE	Outputs a message to the Command Processor System Service console.

≡	SYS_CMD_PRINT	Outputs a printout to the Command Processor System Service console.
≡	SYS_CMD_READY_TO_READ	Allows upper layer application to confirm that the command module is ready to accept command input
≡	SYS_CMD_READY_TO_WRITE	Allows upper layer application to confirm that the command module is ready to write output to the Console System Service.
≡	SYS_CMD_RegisterCallback	Registers a callback function with the command service that will be executed when the lower layer read or write queue is empty.
≡	SYS_CMD_Tasks	Maintains the Command Processor System Service's internal state machine.
≡	SYS_CMDIO_ADD	Adds CMDIO parameters to the Command Processor System Service console.
≡	SYS_CMDIO_GET_HANDLE	Gets the CMDIO Handle base via index.

Macros

	Name	Description
	_promptStr	prompt string
	COMMAND_HISTORY_DEPTH	Command Processor System Service Maximum Depth of Command History.
	ESC_SEQ_SIZE	standard VT100 escape sequences
	LINE_TERM	line terminator
	MAX_CMD_ARGS	Command Processor System Service Maximum Number of Argument definitions.
	MAX_CMD_GROUP	Command Processor System Service Maximum Number of Command Group definitions.
	SYS_CMD_BUFFER_DMA_READY	Define this for MX #define SYS_CMD_BUFFER_DMA_READY __attribute__((coherent)) __attribute__((aligned(4))) //Define this for MZ
	SYS_CMD_DEVICE_MAX_INSTANCES	This is macro SYS_CMD_DEVICE_MAX_INSTANCES.
	SYS_CMD_MAX_LENGTH	Command Processor System Service Command Buffer Maximum Length definition.
	SYS_CMD_MESSAGE	This is macro SYS_CMD_MESSAGE.
	SYS_CMD_PRINT	This is macro SYS_CMD_PRINT.
	SYS_CMD_READ_BUFFER_SIZE	Command Processor System Service Read Buffer Size definition.
	SYS_DEBUG	
	SYS_ERROR	This is macro SYS_ERROR.
	SYS_ERROR_PRINT	This is macro SYS_ERROR_PRINT.

Structures

	Name	Description
◆	SYS_CMD_DEVICE_NODE	Defines the data structure to store each command instance.
	SYS_CMD_API	Identifies the Command API structure.
	SYS_CMD_DESCRIPTOR	a simple command descriptor
	SYS_CMD_DESCRIPTOR_TABLE	table containing the supported commands
	SYS_CMD_DEVICE_LIST	Defines the list structure to store a list of command instances.
	SYS_CMD_INIT	Identifies the system command initialize structure.
	SYS_CMD_INIT_DATA	Defines the data structure to store each command.

Types

	Name	Description
	SYS_CMD_CallbackFunction	Command Callback Function Handle.
	SYS_CMD_DATA_RDY_FNC	Identifies a data available function API.
	SYS_CMD_FNC	Identifies the command process function API.
	SYS_CMD_GETC_FNC	Identifies a get single character function API.
	SYS_CMD_HANDLE	Identifies a particular Command I/O instance.
	SYS_CMD_MSG_FNC	Identifies a message function API.
	SYS_CMD_PRINT_FNC	Identifies a print function API.

	SYS_CMD_PUTC_FNC	Identifies a single character print function API.
	SYS_CMD_READC_FNC	Identifies a read single character function API.

Description

Command Processor System Service Interface Definition

This file contains the interface definition for the Command Processor System Service. It provides a way to interact with the Command Processor subsystem to manage the ASCII command requests from the user supported by the system.

File Name

`sys_command.h`

Company

Microchip Technology Inc.

Console System Service Library

This section describes the Console System Service Library.

Introduction

The Console System Service routes data or message traffic between a console device and a middleware layer or application. The most common use of the Console Service is to route debug or error messages to a terminal program on a host development system.

Description

The Console System Service consists of a core layer and a console device layer. The core layer handles module initialization and system functions. It provides a common API for use by applications and middleware libraries to send and receive data to and from one or more console devices. The Console System core layer maintains the status and device descriptor information for each console instance that has been initialized. Currently, a single instance of console is supported.

The console device layer contains the functional implementation of the core layer APIs. Each console device may have a different implementation, but the behavior of the API should be uniform across different devices. The console device layer interacts with the peripheral libraries (PLIBs) to send and receive data to and from the hardware peripheral. The details of the data transaction are abstracted by the console device layer.

The implementation is non-blocking in nature. The request from the middleware or application return immediately and the data transfer is controlled by a state machine implemented in the console device layer. The calling program is notified of the data transfer completion by a callback mechanism, or can optionally poll the status of the console device layer to see if it is busy (one or more requests are pending) or ready for more data (no request is pending).

The console device layer provides the ability to queue data buffers written to the console. The size of the queue is defined by a configuration option.

Using the Library

This topic describes the basic architecture of the Console System Service Library and provides information and examples on its use.

Description

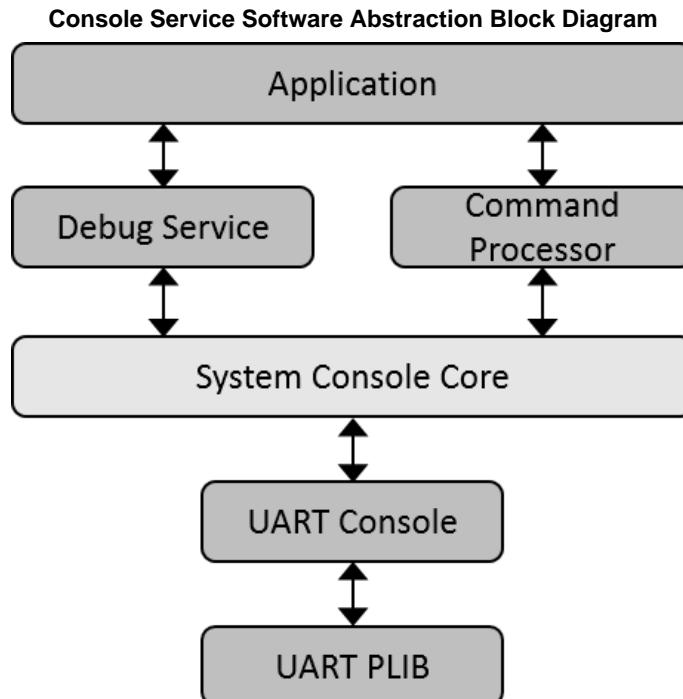
The Console System Service allows the application/middleware to route messages/debug information to a console running on a host computer.

- Depending on the application need, the size of the read and write queues can be configured to allow queuing of multiple requests
- The read/write API called by the calling program returns immediately. The request is queued internally by the console device layer and the data is transferred by the state machine implemented by the console device layer
- The application can register callback with the Console System Service to receive read/write completion/error related notifications

Abstraction Model

This library provides a set of functions that send and receive data to and from a console I/O device.

Description



The Console System Service is a middleware library that is part of a layered software architecture. The purpose of this software layering is to allow each module to provide a consistent interface to its client, thus making it easy for the developer to add and remove modules as needed. The console core layer provides a POSIX-like read/write API to applications and middleware libraries to send and receive data to and from the console device. Data and message buffering, along with the details of the data transfer are contained within the console device and peripheral library (PLIB) layers.

Additional APIs are provided to handle system functions, register a callback function for non-blocking implementations, and flush (reset) the internal queues in the event of an error condition.

How the Library Works

This topic describes the basic architecture of the Console System Service Library and provides information on its implementation.

Description

Each supported console device has its own implementation of the Console System service APIs. Function pointers to the console device implementations are provided in the console device descriptor registered to each console instance. This allows the user to utilize the same API for all console devices, making application code more portable across different devices.

Note: Currently only UART Console Device is implemented.

UART Console Device

The UART Console Device provides a convenient user I/O interface to applications running on a SAM/PIC32 with UART support.

Description

Read

While the user may request multiple bytes of data per read, a typical terminal emulator program sends only one byte per transaction. When multiple bytes are requested, the UART Console Device layer will queue individual byte reads until all bytes are

read. The bytes are sent to the user-supplied read buffer as they are read, and the read callback is triggered when all bytes have been read. Only one read operation can be active at a time. The size of the read queue is a configuration option. The size of the read queue determines the number of read requests (not the number of bytes) that can be queued. Each read request in turn can request for multiple bytes to be read. A call to the console read API with the read queue full returns 0.

The read function accepts a pointer to a read buffer as one of the arguments. It is the responsibility of the application programmer to properly allocate the read buffer.

Write

The UART console device is capable of writing multiple data bytes per transaction, and queuing multiple write transactions. The size of the write queue is a configuration option. If the user tries to queue more write transactions than the write queue can accommodate, the write function will return 0.

The write function accepts a pointer to a write buffer as one of the arguments. It is the responsibility of the application programmer to properly allocate the write buffer.

State Machine

The read and write state machines are run from the interrupt context (from the callback called by the underlying UART PLIB). The underlying PLIB calls the UART console device event handler when the requested number of bytes are transmitted or received (or in case of an error condition). The read/write state machine dispatches application/middleware callback functions and initiates a data transfer if the read/write queue has any request to service.

Register Callback

The callback registration mechanism associates a user function with a read or write completion event. The callback function includes an argument containing a pointer to the read or write buffer associated with the transfer that initiated the callback.

Separate callback functions must be registered for read and write events. If there is no callback function registered, the event will complete as normal, with no callback executed.

Flush

The flush routine in the UART implementation resets the read and write queues. A flush operation is typically triggered in response to an error condition.

Configuring the Library

This section provides information on how to configure the Console System Service library.

Description

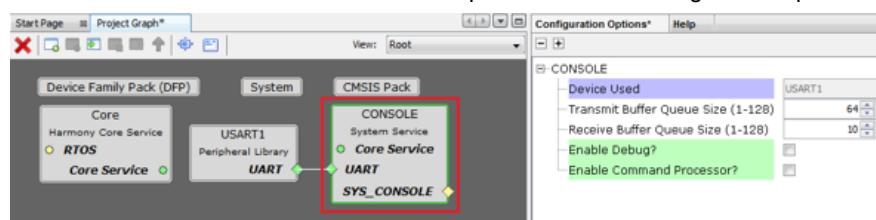
Refer the console device specific configuration options.

UART Console Device Configuration Options

This topic provided configuration option examples for the UART Console Device.

Description

The UART Console System Service library should be configured via MHC. Below is the snapshot of the MHC configuration window for configuring the UART Console Device and a brief description of various configuration options.



Configuration Options:

- Device Used** - Indicates the hardware UART Peripheral Library instance used by the Console System Service.
- Transmit Buffer Queue Size (1-128)** - Indicates the maximum number of data transmit requests that can be queued
- Receive Buffer Queue Size (1-128)** - Indicates the maximum number of data reception requests that can be queued
- Enable Debug?** - Enable Debug System Service. Refer the Debug System Service for more details on configuring the Debug service.

- **Enable Command Processor?** - Enable Command Processor System Service. Refer the Command Processor System Service for more details on configuring the Command Processor System service.

Building the Library

This section provides information on how the Console System Service Library can be built.

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Functions

	Name	Description
≡	SYS_CONSOLE_Initialize	Initializes the console instance module and opens or initializes the specific module instance to which it is associated.
≡	SYS_CONSOLE_Status	Returns status of the specific instance of the Console module.
≡	SYS_CONSOLE_Tasks	Maintains the console's state machine.

b) Core Functions

	Name	Description
≡	SYS_CONSOLE_Read	Reads data from the console device.
≡	SYS_CONSOLE_Write	Writes data to the console device.
≡	SYS_CONSOLE_RegisterCallback	Registers a callback function with the console service that will be executed when the read or write request is complete (or in case of an error condition during read/write).
≡	SYS_CONSOLE_Flush	Flushes the read and write queues for the given console instance.

c) Data Types and Constants

	Name	Description
	STDERR_FILENO	This is macro STDERR_FILENO.
	STDIN_FILENO	These are in unistd.h
	STDOUT_FILENO	This is macro STDOUT_FILENO.
	SYS_CONSOLE_INDEX_0	Console System Service index definitions.
	SYS_CONSOLE_INDEX_1	This is macro SYS_CONSOLE_INDEX_1.
	SYS_CONSOLE_INDEX_2	This is macro SYS_CONSOLE_INDEX_2.
	SYS_CONSOLE_INDEX_3	This is macro SYS_CONSOLE_INDEX_3.
	SYS_CONSOLE_MESSAGE	This is macro SYS_CONSOLE_MESSAGE.
	SYS_CONSOLE_PRINT	This is macro SYS_CONSOLE_PRINT.
	SYS_CONSOLE_CALLBACK	Pointer to a console system service callback function.
	SYS_CONSOLE_EVENT	Identifies the system console event for which the callback is being registered.

Description

This section describes the APIs of the Console System Service Library.

Refer to each section for a detailed description.

a) System Functions

SYS_CONSOLE_Initialize Function

Initializes the console instance module and opens or initializes the specific module instance to which it is associated.

File

[sys_console.h](#)

C

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

Returns

If successful, returns a valid handle to the console instance. Otherwise, it returns [SYS_MODULE_OBJ_INVALID](#). The returned object must be passed as argument to [SYS_CONSOLE_Tasks](#) and [SYS_CONSOLE_Status](#) routines.

Description

This function initializes the internal data structures used by the console module. It also initializes the associated I/O driver/PLIB.

Remarks

This routine should only be called once during system initialization.

Preconditions

None.

Example

```
SYS_MODULE_OBJ objectHandle;  
  
// Populate the console initialization structure  
const SYS_CONSOLE_INIT sysConsole0Init =  
{  
    .deviceInitData = (void*)&sysConsole0UARTInitData,  
    .consDevDesc = &sysConsole0UARTDevDesc,  
    .deviceIndex = 0,  
};  
  
objectHandle = SYS_CONSOLE_Initialize(SYS_CONSOLE_INDEX_0, (SYS_MODULE_INIT *) &sysConsole0Init);  
if (objectHandle == SYS_MODULE_OBJ_INVALID)  
{  
    // Handle error  
}
```

Parameters

Parameters	Description
index	Index for the instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the Console System service. This pointer may be null if no data is required because static overrides have been provided.

Function

```
SYS_MODULE_OBJ SYS_CONSOLE_Initialize(  
    const SYS_MODULE_INDEX index,  
    const SYS_MODULE_INIT* const init  
)
```

SYS_CONSOLE_Status Function

Returns status of the specific instance of the Console module.

File

[sys_console.h](#)

C

```
SYS_STATUS SYS_CONSOLE_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that the driver is initialized and is ready to accept new requests from the clients.
- SYS_STATUS_BUSY - Indicates that the driver is busy with a previous requests from the clients. However, depending on the configured queue size for transmit and receive, it may be able to queue a new request.
- 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_STATUS_UNINITIALIZED - Indicates that the driver is not initialized.

Description

This function returns the status of the specific module instance.

Remarks

None.

Preconditions

The [SYS_CONSOLE_Initialize](#) function should have been called before calling this function.

Example

```
// Given "object" returned from SYS_CONSOLE_Initialize

SYS_STATUS consStatus;

consStatus = SYS_CONSOLE_Status (object);
if (consStatus == SYS_STATUS_READY)
{
    // Console is initialized and is ready to accept client requests.
}
```

Parameters

Parameters	Description
object	SYS CONSOLE object handle, returned from SYS_CONSOLE_Initialize

Function

[SYS_STATUS SYS_CONSOLE_Status \(SYS_MODULE_OBJ object \)](#)

SYS_CONSOLE_Tasks Function

Maintains the console's state machine.

File

[sys_console.h](#)

C

```
void SYS_CONSOLE_Tasks(SYS_MODULE_OBJ object);
```

Returns

None

Description

This function runs the console system service's internal state machine.

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.

Preconditions

The [SYS_CONSOLE_Initialize](#) function must have been called for the specified CONSOLE driver instance.

Example

```
SYS_MODULE_OBJ object;           // Returned from SYS_CONSOLE_Initialize

while (true)
{
    SYS_CONSOLE_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	SYS CONSOLE object handle, returned from SYS_CONSOLE_Initialize

Function

```
void SYS_CONSOLE_Tasks ( SYS_MODULE_OBJ object )
```

b) Core Functions**SYS_CONSOLE_Read Function**

Reads data from the console device.

File

[sys_console.h](#)

C

```
ssize_t SYS_CONSOLE_Read(const SYS_MODULE_INDEX index, int fd, void* buf, size_t count);
```

Returns

The requested number of bytes to read is returned back if the request is accepted successfully. In case of error, the returned value is less than the requested number of bytes to read.

Description

This function reads the data from the console device.

Remarks

None.

Preconditions

The [SYS_CONSOLE_Initialize](#) function should have been called before calling this function.

Example

```
ssize_t nr;
char myBuffer[MY_BUFFER_SIZE];
nr = SYS_CONSOLE_Read( SYS_CONSOLE_INDEX_0, 0, myBuffer, MY_BUFFER_SIZE );
if (nr != MY_BUFFER_SIZE)
{
    // handle error
}
```

Parameters

Parameters	Description
index	Console instance index
fd	I/O stream handle. Maintained for backward compatibility. NULL value can be passed as a parameter.
buf	Buffer to hold the read data.
count	Number of bytes to read.

Function

```
ssize_t SYS_CONSOLE_Read(
    const SYS_MODULE_INDEX index,
    int fd,
    void* buf,
    size_t count
)
```

SYS_CONSOLE_Write Function

Writes data to the console device.

File

[sys_console.h](#)

C

```
ssize_t SYS_CONSOLE_Write(const SYS_MODULE_INDEX index, int fd, const void* buf, size_t count);
```

Returns

The requested number of bytes to write is returned back if the request is accepted successfully. In case of error, the returned value is less than the requested number of bytes to write.

Description

This function writes data to the console device.

Remarks

None.

Preconditions

The [SYS_CONSOLE_Initialize](#) function should have been called before calling this function.

Example

```
ssize_t nr;
char myBuffer[] = "message";
nr = SYS_CONSOLE_Write( SYS_CONSOLE_INDEX_0, 0, myBuffer, strlen(myBuffer) );
if (nr != strlen(myBuffer))
```

```
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Console instance index
fd	I/O stream handle. Maintained for backward compatibility. NULL value can be passed as a parameter.
buf	Buffer holding the data to be written.
count	Number of bytes to write.

Function

```
ssize_t SYS_CONSOLE_Write(
    const SYS_MODULE_INDEX index,
    int fd,
    const void* buf,
    size_t count
)
```

SYS_CONSOLE_RegisterCallback Function

Registers a callback function with the console service that will be executed when the read or write request is complete (or in case of an error condition during read/write).

File

[sys_console.h](#)

C

```
void SYS_CONSOLE_RegisterCallback(const SYS_MODULE_INDEX index, SYS_CONSOLE_CALLBACK cbFunc,
    SYS_CONSOLE_EVENT event);
```

Returns

None.

Description

This function is used by an application to register a callback function with the console service. The callback function is called in response to a read/write completion (or error) event. Separate callback functions are required for each event. To receive events, the callback must be registered before submitting a read/write request.

Remarks

None.

Preconditions

The [SYS_CONSOLE_Initialize](#) function should have been called before calling this function.

Example

```
//Registering a "APP_ReadComplete" read callback function
SYS_CONSOLE_RegisterCallback(SYS_CONSOLE_INDEX_0, APP_ReadComplete,
    SYS_CONSOLE_EVENT_READ_COMPLETE);

//Registering a "APP_WriteComplete" write callback function
SYS_CONSOLE_RegisterCallback(SYS_CONSOLE_INDEX_0, APP_WriteComplete,
    SYS_CONSOLE_EVENT_WRITE_COMPLETE);
```

Parameters

Parameters	Description
index	Console instance index
cbFunc	The name of the callback function
event	Enumerated list of events that can trigger a callback

Function

```
void SYS_CONSOLE_RegisterCallback(
    const SYS_MODULE_INDEX index,
    SYS_CONSOLE_CALLBACK cbFunc,
    SYS_CONSOLE_EVENT event
)
```

SYS_CONSOLE_Flush Function

Flushes the read and write queues for the given console instance.

File

[sys_console.h](#)

C

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

Returns

None.

Description

This function flushes queued read and write requests. The request that is already in progress cannot be flushed. Only the queued pending requests will be flushed.

Remarks

None.

Preconditions

The [SYS_CONSOLE_Initialize](#) function should have been called before calling this function.

Example

```
SYS_CONSOLE_Flush(SYS_CONSOLE_INDEX_0);
```

Parameters

Parameters	Description
index	Console instance index

Function

```
void SYS_CONSOLE_Flush(const SYS_MODULE_INDEX index)
```

c) Data Types and Constants

STDERR_FILENO Macro

File

[sys_console.h](#)

C

```
#define STDERR_FILENO 2
```

Description

This is macro STDERR_FILENO.

STDIN_FILENO Macro

File

[sys_console.h](#)

C

```
#define STDIN_FILENO 0
```

Description

These are in unistd.h

STDOUT_FILENO Macro

File

[sys_console.h](#)

C

```
#define STDOUT_FILENO 1
```

Description

This is macro STDOUT_FILENO.

SYS_CONSOLE_INDEX_0 Macro

Console System Service index definitions.

File

[sys_console.h](#)

C

```
#define SYS_CONSOLE_INDEX_0 0
```

Description

SYS Console Module Index Numbers

These constants provide Console System Service index definitions.

Remarks

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

SYS_CONSOLE_INDEX_1 Macro

File

[sys_console.h](#)

C

```
#define SYS_CONSOLE_INDEX_1 1
```

Description

This is macro SYS_CONSOLE_INDEX_1.

SYS_CONSOLE_INDEX_2 Macro

File

[sys_console.h](#)

C

```
#define SYS_CONSOLE_INDEX_2 2
```

Description

This is macro SYS_CONSOLE_INDEX_2.

SYS_CONSOLE_INDEX_3 Macro

File

[sys_console.h](#)

C

```
#define SYS_CONSOLE_INDEX_3 3
```

Description

This is macro SYS_CONSOLE_INDEX_3.

SYS_CONSOLE_MESSAGE Macro

File

[sys_console.h](#)

C

```
#define SYS_CONSOLE_MESSAGE(message)
```

Description

This is macro SYS_CONSOLE_MESSAGE.

SYS_CONSOLE_PRINT Macro

File

[sys_console.h](#)

C

```
#define SYS_CONSOLE_PRINT(fmt, ...)
```

Description

This is macro SYS_CONSOLE_PRINT.

SYS_CONSOLE_CALLBACK Type

Pointer to a console system service callback function.

File

[sys_console.h](#)

C

```
typedef void (* SYS_CONSOLE_CALLBACK)(void* pBuffer);
```

Returns

None.

Description

This data type defines a pointer to a console service callback function, thus defining the function signature.

Remarks

None.

Preconditions

The console service must have been initialized using the [SYS_CONSOLE_Initialize](#) function before attempting to register a SYS Console callback function.

Example

```
void MyCallback ( void* pBuffer )
{
    if (pBuffer != NULL)
    {
        //Free the memory pointed by pBuffer if it was allocated dynamically.
    }
}
```

Parameters

Parameters	Description
pBuffer	Pointer to the processed read/write buffer. It can be used identify the buffer that is processed by the console system service and free the buffer memory if it was allocated dynamically.

Function

```
void ( * SYS_CONSOLE_CALLBACK ) ( void* pBuffer )
```

SYS_CONSOLE_EVENT Enumeration

Identifies the system console event for which the callback is being registered.

File

[sys_console.h](#)

C

```
typedef enum {
```

```

SYS_CONSOLE_EVENT_WRITE_COMPLETE,
SYS_CONSOLE_EVENT_READ_COMPLETE
} SYS_CONSOLE_EVENT;

```

Members

Members	Description
SYS_CONSOLE_EVENT_WRITE_COMPLETE	System console write complete event
SYS_CONSOLE_EVENT_READ_COMPLETE	System console read complete event

Description

System Console Event

This enum is used to identify if the callback being registered is to be called on a read or on a write complete event.

Files

Files

Name	Description
sys_console.h	Console System Service interface definitions.

Description

This section lists the header files used by the library.

sys_console.h

Console System Service interface definitions.

Enumerations

	Name	Description
	SYS_CONSOLE_EVENT	Identifies the system console event for which the callback is being registered.

Functions

	Name	Description
≡	SYS_CONSOLE_Flush	Flushes the read and write queues for the given console instance.
≡	SYS_CONSOLE_Initialize	Initializes the console instance module and opens or initializes the specific module instance to which it is associated.
≡	SYS_CONSOLE_Read	Reads data from the console device.
≡	SYS_CONSOLE_RegisterCallback	Registers a callback function with the console service that will be executed when the read or write request is complete (or in case of an error condition during read/write).
≡	SYS_CONSOLE_Status	Returns status of the specific instance of the Console module.
≡	SYS_CONSOLE_Tasks	Maintains the console's state machine.
≡	SYS_CONSOLE_Write	Writes data to the console device.

Macros

	Name	Description
	STDERR_FILENO	This is macro STDERR_FILENO.
	STDIN_FILENO	These are in unistd.h
	STDOUT_FILENO	This is macro STDOUT_FILENO.
	SYS_CONSOLE_INDEX_0	Console System Service index definitions.
	SYS_CONSOLE_INDEX_1	This is macro SYS_CONSOLE_INDEX_1.
	SYS_CONSOLE_INDEX_2	This is macro SYS_CONSOLE_INDEX_2.
	SYS_CONSOLE_INDEX_3	This is macro SYS_CONSOLE_INDEX_3.

	SYS_CONSOLE_MESSAGE	This is macro SYS_CONSOLE_MESSAGE.
	SYS_CONSOLE_PRINT	This is macro SYS_CONSOLE_PRINT.

Types

	Name	Description
	SYS_CONSOLE_CALLBACK	Pointer to a console system service callback function.

Description

Console System Service Interface Definition

This file contains the interface definition for the Console system service. It provides a way to interact with the Console subsystem to manage the timing requests supported by the system.

File Name

sys_console.h

Company

Microchip Technology Inc.

Debug System Service Library

This section describes the Debug System Service Library.

Introduction

This library implements the Debug System Service. The Debug System Service provides a convenient mechanism for the application developer to send formatted or unformatted messages to a system console.

Description

The Debug System Service provides a set of functions that allow the developer to output debug and/or error messages based on a global error level. The messages are sent to the System Console Service, where they are routed to a console device.

The Debug System Service maintains a global error level, which may be set during initialization and changed dynamically at run-time. Both formatted and unformatted messages can be output or not, depending on the current global error level.

Using the Library

This topic describes the basic architecture of the Debug System Service Library and provides information on its use.

Description

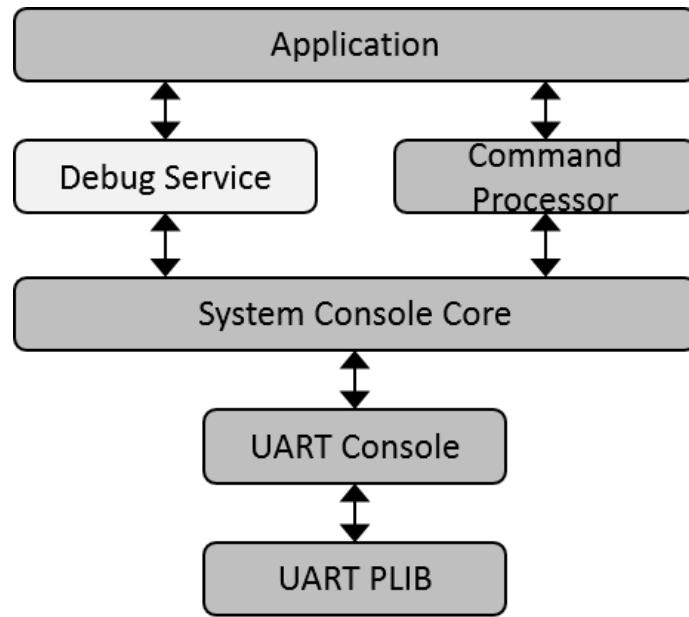
When the Debug System Service is initialized, it sets the global system error level to the specified level. This level determines the threshold at which debug and error messages are sent to the console. This allows different debug and error reporting verbosity depending on the needs of the developer. The Debug System Service also provides APIs to dynamically set the error level during program execution. This allows the developer to increase or decrease debug verbosity to specific areas of interest within the program.

Abstraction Model

This library provides a set of functions that allow the developer to quickly and easily provide debug and error messaging during and after program development.

Description

Debug Service Software Abstraction Block Diagram



The Debug System Service provides APIs for the application developer to send formatted or unformatted messages to a system console. The console core layer provides a POSIX-like read/write API to applications and middleware libraries to send and receive data to and from the console device. Data and message buffering, along with the details of the data transfer are contained within the console device and peripheral library (PLIB) layers

How the Library Works

The Debug System Service library can be used by a device driver, middleware layer, or application to report error conditions and output debug messages to a console device during program operation.

Description

Debug Messages and Error Reporting

The following macros are available to output debug and error messages. The default implementation of these macros resolves to nothing by the preprocessor. This is to allow the developer to leave debug messaging in the released code without impacting code size or performance. Typically, the developer would define `SYS_DEBUG_USE_CONSOLE` macro through MHC configuration for debug builds, to map these macros to appropriate debug service implementation.

- `SYS_MESSAGE(message)` prints a simple message string to the output device irrespective of the value of the global error level.
- `SYS_DEBUG_MESSAGE(level, message)` prints a debug message to the console device if the global error level is equal to or lower than that specified by the "level" argument.
- `SYS_PRINT(fmt, ...)` prints formatted messages to the console. The message formatting is the same as `printf`.
- `SYS_DEBUG_PRINT(level, fmt, ...)` prints formatted messages to the console if the global error level is equal to or lower than that specified by the "level" argument. The message formatting is the same as `printf`.

Changing the System Error Level

Two functions are provided to manipulate the global system error level at runtime. This may be useful when you want to increase the debug verbosity for a particular section of code, but not for the entire program.

- `SYS_DEBUG_ErrorLevelGet()` returns the current global system error level
- `SYS_DEBUG_ErrorLevelSet(level)` sets the current global system error level

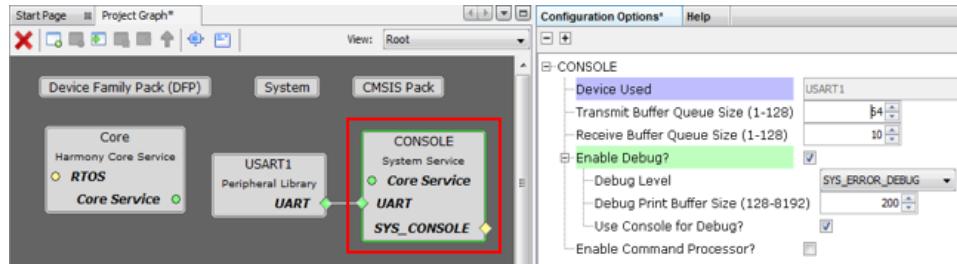
Configuring the Library

This section provides information on how to configure the Debug System Service library.

Description

The Debug System Service library should be configured via MHC. Below is the snapshot of the MHC configuration window and a brief description of various configuration options.

In MHC, the Debug System Service configuration options are available under the Console System Service component.



Configuration Options (See Note below):

- Device Used** - Indicates the hardware UART Peripheral Library instance used by the Console System Service.
- Transmit Buffer Queue Size (1-128)** - Indicates the maximum number of data transmit requests that can be queued
- Receive Buffer Queue Size (1-128)** - Indicates the maximum number of data reception requests that can be queued
- Enable Debug?** - Enable Debug System Service
 - Debug Level** - Indicates the global error level set during initialization. Application can change the global error level during runtime using an API
 - Debug Print Buffer Size (128-8192)** - Indicates the size of the print buffer in bytes
 - Use Console for Debug?** Check this option to map the debug macros to debug implementation. Un-checking this option maps the debug macros to nothing
- Enable Command Processor?** - Enable Command Processor System Service. Refer the Command Processor System Service for more information on configuring the Command Processor System Service.

Building the Library

This section provides information on how the Debug System Service Library can be built.

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Functions

	Name	Description
≡	SYS_DEBUG_Initialize	Initializes the global error level and specific module instance.
≡	SYS_DEBUG_Status	Returns status of the specific instance of the debug service module.
≡	SYS_DEBUG_Tasks	Maintains the debug module's state machine.
≡	SYS_DEBUG_Message	Prints a message to the console regardless of the system error level.
≡	SYS_DEBUG_Print	Formats and prints a message with a variable number of arguments to the console regardless of the system error level.
≡	SYS_DEBUG_BreakPoint	Inserts a software breakpoint instruction when building in Debug mode.

b) Changing System Error Level Functions

	Name	Description
≡	SYS_DEBUG_ErrorLevelGet	Returns the global system Error reporting level.

	SYS_DEBUG_ErrorLevelSet	Sets the global system error reporting level.
---	---	---

c) Data Types and Constants

	Name	Description
	SYS_ERROR_LEVEL	System error message priority levels.
	SYS_DEBUG_INDEX_0	Debug System Service index.
	_SYS_DEBUG_MESSAGE	Prints a debug message if the specified level is at or below the global system error level.
	_SYS_DEBUG_PRINT	Formats and prints a debug message if the specified level is at or below the global system error level.

Description

This section describes the APIs of the Debug System Service Library.

Refer to each section for a detailed description.

a) System Functions

SYS_DEBUG_Initialize Function

Initializes the global error level and specific module instance.

File

[sys_debug.h](#)

C

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

Returns

If successful, returns [SYS_MODULE_OBJ_STATIC](#). Otherwise, it returns [SYS_MODULE_OBJ_INVALID](#).

Description

This function initializes the global error level. It also initializes any internal system debug module data structures.

Remarks

This routine should only be called once during system initialization.

Preconditions

None.

Example

```
SYS_MODULE_OBJ objectHandle;
SYS_DEBUG_INIT debugInit =
{
    .moduleInit = {0},
    .errorLevel = SYS_DEBUG_GLOBAL_ERROR_LEVEL,
    .consoleIndex = 0,
};

objectHandle = SYS_DEBUG_Initialize(SYS_DEBUG_INDEX_0, (SYS_MODULE_INIT*)&debugInit);
if (objectHandle == SYS_MODULE_OBJ_INVALID)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Index for the instance to be initialized.
init	Pointer to a data structure containing any data necessary to initialize the debug service. This pointer may be null if no data is required because static overrides have been provided.

Function

```
SYS_MODULE_OBJ SYS_DEBUG_Initialize(
    const SYS_MODULE_INDEX index,
    const SYS_MODULE_INIT* const init
)
```

SYS_DEBUG_Status Function

Returns status of the specific instance of the debug service module.

File

[sys_debug.h](#)

C

```
SYS_STATUS SYS_DEBUG_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that the module is running and ready to service requests. Any value greater than SYS_STATUS_READY is also a normal running state in which the module is ready to accept new operations.
- SYS_STATUS_BUSY - Indicates that the module is busy with a previous system level operation.
- SYS_STATUS_ERROR - Indicates that the module is in an error state. Any value less than SYS_STATUS_ERROR is also an error state.
- SYS_STATUS_UNINITIALIZED - Indicates that the module has not been initialized.

Description

This function returns the status of the specific debug service module instance.

Remarks

None.

Preconditions

The [SYS_DEBUG_Initialize](#) function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ          object;      // Returned from SYS_CONSOLE_Initialize
SYS_STATUS              debugStatus;

debugStatus = SYS_DEBUG_Status (object);
if (debugStatus == SYS_STATUS_READY)
{
    // Debug service is initialized and ready to accept requests.
}
```

Parameters

Parameters	Description
object	Debug module object handle, returned from SYS_DEBUG_Initialize

Function

```
SYS_STATUS SYS_DEBUG_Status ( SYS_MODULE_OBJ object )
```

SYS_DEBUG_Tasks Function

Maintains the debug module's state machine.

File

[sys_debug.h](#)

C

```
void SYS_DEBUG_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the debug module's internal state machine.

Remarks

This function is normally not called directly by an application. The task routine may not be called if the debug service does not require maintaining an internal state machine.

Preconditions

The [SYS_DEBUG_Initialize](#) function must have been called.

Example

```
SYS_MODULE_OBJ object;      // Returned from SYS_DEBUG_Initialize

while (true)
{
    SYS_DEBUG_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	SYS DEBUG object handle, returned from SYS_DEBUG_Initialize

Function

```
void SYS_DEBUG_Tasks( SYS_MODULE_OBJ object )
```

SYS_DEBUG_Message Function

Prints a message to the console regardless of the system error level.

File

[sys_debug.h](#)

C

```
void SYS_DEBUG_Message(const char * message);
```

Returns

None.

Description

This function prints a message to the console regardless of the system error level. It can be used as an implementation of the [SYS_MESSAGE](#) and [SYS_DEBUG_MESSAGE](#) macros.

Remarks

Do not call this function directly. Call the [SYS_MESSAGE](#) or [SYS_DEBUG_MESSAGE](#) macros instead.

The default [SYS_MESSAGE](#) and [SYS_DEBUG_MESSAGE](#) macro definitions remove the messages and message function calls from the source code. To access and utilize the messages, define the [SYS_DEBUG_USE_CONSOLE](#) macro or override the definitions of the individual macros.

Preconditions

[SYS_DEBUG_Initialize](#) must have returned a valid object handle.

Example

```
// In configuration.h file: #define SYS_DEBUG_USE_CONSOLE
// In sys_debug.h file: #define SYS_MESSAGE(message) SYS_DEBUG_Message(message)

SYS_MESSAGE( "My Messagern" );
```

Parameters

Parameters	Description
message	Pointer to a message string to be displayed.

Function

`void SYS_DEBUG_Message(const char *message)`

SYS_DEBUG_Print Function

Formats and prints a message with a variable number of arguments to the console regardless of the system error level.

File

[sys_debug.h](#)

C

```
void SYS_DEBUG_Print(const char * format, ...);
```

Returns

None.

Description

This function formats and prints a message with a variable number of arguments to the console regardless of the system error level. It can be used to implement the [SYS_PRINT](#) and [SYS_DEBUG_PRINT](#) macros.

Remarks

The format string and arguments follow the printf convention.

Do not call this function directly. Call the [SYS_PRINT](#) or [SYS_DEBUG_PRINT](#) macros instead.

The default [SYS_PRINT](#) and [SYS_DEBUG_PRINT](#) macro definitions remove the messages and message function calls. To access and utilize the messages, define the [SYS_DEBUG_USE_CONSOLE](#) macro or override the definitions of the individual macros.

Preconditions

[SYS_DEBUG_Initialize](#) must have returned a valid object handle.

Example

```
// In configuration.h file: #define SYS_DEBUG_USE_CONSOLE
// In sys_debug.h file: #define SYS_PRINT(fmt, ...) SYS_DEBUG_Print(fmt, ##__VA_ARGS__)

// In source code
int result;

result = SomeOperation();
```

```

if (result > MAX_VALUE)
{
    SYS_PRINT("Result of %d exceeds max value", result);
}

```

Parameters

Parameters	Description
format	Pointer to a buffer containing the format string for the message to be displayed.
...	Zero or more optional parameters to be formatted as defined by the format string.

Function

```
void SYS_DEBUG_Print(const char *format, ...)
```

SYS_DEBUG_BreakPoint Macro

Inserts a software breakpoint instruction when building in Debug mode.

File

[sys_debug.h](#)

C

```
#define SYS_DEBUG_BreakPoint
```

Returns

None.

Description

Macro: SYS_DEBUG_BreakPoint(void)

This macro inserts a software breakpoint instruction when building in Debug mode.

Remarks

Compiles out if not built for debugging.

Preconditions

None.

Example

```

if (myDebugTestFailed)
{
    SYS_DEBUG_BreakPoint();
}

```

b) Changing System Error Level Functions

SYS_DEBUG_ErrorLevelGet Function

Returns the global system Error reporting level.

File

[sys_debug.h](#)

C

```
SYS_ERROR_LEVEL SYS_DEBUG_ErrorLevelGet();
```

Returns

The global System Error Level.

Description

This function returns the global System Error reporting level.

Remarks

None.

Preconditions

`SYS_DEBUG_Initialize` must have returned a valid object handle.

Example

```
SYS_ERROR_LEVEL level;  
  
level = SYS_DEBUG_ErrorLevelGet();
```

Function

`SYS_ERROR_LEVEL SYS_DEBUG_ErrorLevelGet(void)`

SYS_DEBUG_ErrorLevelSet Function

Sets the global system error reporting level.

File

`sys_debug.h`

C

```
void SYS_DEBUG_ErrorLevelSet(SYS_ERROR_LEVEL level);
```

Returns

None.

Description

This function sets the global system error reporting level.

Remarks

None.

Preconditions

`SYS_DEBUG_Initialize` must have returned a valid object handle.

Example

```
SYS_DEBUG_ErrorLevelSet(SYS_ERROR_WARNING);
```

Parameters

Parameters	Description
level	The desired system error level.

Function

```
void SYS_DEBUG_ErrorLevelSet( SYS_ERROR_LEVEL level)
```

c) Data Types and Constants

SYS_ERROR_LEVEL Enumeration

System error message priority levels.

File

[sys_debug.h](#)

C

```
typedef enum {
    SYS_ERROR_FATAL = 0,
    SYS_ERROR_ERROR = 1,
    SYS_ERROR_WARNING = 2,
    SYS_ERROR_INFO = 3,
    SYS_ERROR_DEBUG = 4
} SYS_ERROR_LEVEL;
```

Members

Members	Description
SYS_ERROR_FATAL = 0	Errors that have the potential to cause a system crash.
SYS_ERROR_ERROR = 1	Errors that have the potential to cause incorrect behavior.
SYS_ERROR_WARNING = 2	Warnings about potentially unexpected behavior or side effects.
SYS_ERROR_INFO = 3	Information helpful to understanding potential errors and warnings.
SYS_ERROR_DEBUG = 4	Verbose information helpful during debugging and testing.

Description

SYS_ERROR_LEVEL enumeration

This enumeration defines the supported system error message priority values.

Remarks

Used by debug message macros to compare individual message priority against a global system-wide error message priority level to determine if an individual message should be displayed.

SYS_DEBUG_INDEX_0 Macro

Debug System Service index.

File

[sys_debug.h](#)

C

```
#define SYS_DEBUG_INDEX_0 0
```

Description

SYS Debug Module Index Number

This constant defines a symbolic name for the debug system service index.

Remarks

There can only be a single debug system service instance in the system.

SYS_DEBUG_MESSAGE Macro

Prints a debug message if the specified level is at or below the global system error level.

File[sys_debug.h](#)**C**

```
#define _SYS_DEBUG_MESSAGE(level, message) do { if((level) <= SYS_DEBUG_ErrorLevelGet())  
SYS_DEBUG_Message(message); }while(0)
```

Returns

None.

DescriptionMacro: `_SYS_DEBUG_MESSAGE(SYS_ERROR_LEVEL level, const char* message)`

This macro prints a debug message if the specified level is at or below the global error level. It can be used to implement the [SYS_DEBUG_MESSAGE](#) macro.

Remarks

Do not call this macro directly. Call the [SYS_DEBUG_MESSAGE](#) macro instead.

The default [SYS_DEBUG_MESSAGE](#) macro definition removes the message and function call from the source code. To access and utilize the message, define the [SYS_DEBUG_USE_CONSOLE](#) macro or override the definition of the [SYS_DEBUG_MESSAGE](#) macro.

Preconditions

[SYS_DEBUG_Initialize](#) must have returned a valid object handle.

Example

```
// In configuration.h file: #define SYS_DEBUG_USE_CONSOLE  
// In sys_debug.h file: #define SYS_DEBUG_MESSAGE( level,message )  
_SYS_DEBUG_MESSAGE( level,message )  
  
// In source code  
SYS_DEBUG_MESSAGE(SYS_ERROR_WARNING, "My debug warning messagern");
```

Parameters

Parameters	Description
level	The current error level threshold for displaying the message.
message	Pointer to a buffer containing the message to be displayed.

_SYS_DEBUG_PRINT Macro

Formats and prints a debug message if the specified level is at or below the global system error level.

File[sys_debug.h](#)**C**

```
#define _SYS_DEBUG_PRINT(level, format, ...) do { if((level) <= SYS_DEBUG_ErrorLevelGet())  
SYS_DEBUG_Print(format, ##__VA_ARGS__); } while (0)
```

Returns

None.

DescriptionMacro: `_SYS_DEBUG_PRINT(SYS_ERROR_LEVEL level, const char* format, ...)`

This function formats and prints a debug message if the specified level is at or below the global system error level. It can be used to implement the [SYS_DEBUG_PRINT](#) macro.

Remarks

Do not call this macro directly. Call the [SYS_DEBUG_PRINT](#) macro instead.

The default [SYS_DEBUG_PRINT](#) macro definition removes the message and function call from the source code. To access and utilize the message, define the [SYS_DEBUG_USE_CONSOLE](#) macro or override the definition of the [SYS_DEBUG_PRINT](#) macro.

Preconditions

[SYS_DEBUG_Initialize](#) must have returned a valid object handle.

Example

```
// In configuration.h file: #define SYS_DEBUG_USE_CONSOLE
// In sys_debug.h file: #define SYS_DEBUG_PRINT(level, fmt, ...) _SYS_DEBUG_PRINT(level, fmt,
##__VA_ARGS__)

// In source code
int result;

result = SomeOperation();
if (result > MAX_VALUE)
{
    SYS_DEBUG_PRINT(SYS_ERROR_WARNING, "Result of %d exceeds max value", result);
    // Take appropriate action
}
```

Parameters

Parameters	Description
level	The current error level threshold for displaying the message.
format	Pointer to a buffer containing the format string for the message to be displayed.
...	Zero or more optional parameters to be formatted as defined by the format string.

Files

Files

Name	Description
sys_debug.h	Defines the common debug definitions and interfaces used by MPLAB Harmony libraries to report errors and debug information to the user.

Description

This section lists the header files used by the library.

sys_debug.h

Defines the common debug definitions and interfaces used by MPLAB Harmony libraries to report errors and debug information to the user.

Enumerations

	Name	Description
	SYS_ERROR_LEVEL	System error message priority levels.

Functions

	Name	Description
≡	SYS_DEBUG_ErrorLevelGet	Returns the global system Error reporting level.
≡	SYS_DEBUG_ErrorLevelSet	Sets the global system error reporting level.

≡	SYS_DEBUG_Initialize	Initializes the global error level and specific module instance.
≡	SYS_DEBUG_Message	Prints a message to the console regardless of the system error level.
≡	SYS_DEBUG_Print	Formats and prints a message with a variable number of arguments to the console regardless of the system error level.
≡	SYS_DEBUG_Status	Returns status of the specific instance of the debug service module.
≡	SYS_DEBUG_Tasks	Maintains the debug module's state machine.

Macros

	Name	Description
	_SYS_DEBUG_MESSAGE	Prints a debug message if the specified level is at or below the global system error level.
	_SYS_DEBUG_PRINT	Formats and prints a debug message if the specified level is at or below the global system error level.
	SYS_DEBUG_BreakPoint	Inserts a software breakpoint instruction when building in Debug mode.
	SYS_DEBUG_INDEX_0	Debug System Service index.
	SYS_DEBUG_MESSAGE	Prints a debug message if the system error level is defined at or lower than the level specified.
	SYS_DEBUG_PRINT	Formats and prints an error message if the system error level is defined at or lower than the level specified.
	SYS_MESSAGE	Prints a message to the console regardless of the system error level.
	SYS_PRINT	Formats and prints an error message with a variable number of arguments regardless of the system error level.

Description

Debug System Services Library Header

This header file defines the common debug definitions and interface macros (summary below) and prototypes used by MPLAB Harmony libraries to report errors and debug information to the user.

File Name

sys_debug.h

Company

Microchip Technology Inc.

Direct Memory Access (DMA) System Service Library

Introduction

This library provides an abstracted interface to interact with the DMA subsystem to control and manage the data transfer between different peripherals and/or memory without intervention from the CPU.

Description

The Direct Memory Access (DMA) controller is a bus master module that is useful for data transfers between different peripherals without intervention from the CPU. The source and destination of a DMA transfer can be any of the memory-mapped modules. For example, memory, or one of the Peripheral Bus (PBUS) devices such as the SPI, UART, and so on.

This library provides a low-level abstraction of the DMA System Service Library that is available on the Microchip family of PIC32 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/PLIB, thereby hiding differences from one microcontroller variant to another.

Using the Library

This topic describes the basic architecture of the DMA System Service Library and provides information and examples on its use.

Description

Interface Header File: `sys_dma.h`

The interface to the DMA System Service library is defined in the `sys_dma.h` header file. This file is included by the `system.h` file. Any C language source (.c) file that uses the DMA System Service Library should include `system.h`.

Please refer to the [What is MPLAB Harmony?](#) section for how the DMA System Service interacts with the framework.

Abstraction Model

This model explains how the system interfaces with the DMA System Service and the application.

Description

This library provides an abstraction of the DMA subsystem that is used by device drivers, middleware libraries and applications to transmit and receive data.



DMA System Service

The DMA system services provide support for initializing the DMA controller, managing the transfer state machine, setup/management of channels, and global module control and status management.

Initialization and Tasks

Each software module (device driver, middleware, or application) that needs to use the DMA for data transfer must enable the DMA controller. This is normally done by calling the initialization routine of the DMA subsystem in the module's initialization routine, which is called by the `SYS_Initialize` service. The initialization routine returns a DMA module object, which should be used as a parameter in the call to Task routines.

The Task routines implement the data transfer state machine for synchronous and asynchronous data transfer operations. If Asynchronous (interrupt) mode of operation is desired, the Task routine (`SYS_DMA_Tasks`) should be called from the respective channel ISR. If Synchronous mode of operation is desired, the Task routine (`SYS_DMA_Tasks`) should be called from the `SYS_Tasks` function.

Channel Setup and Management

Any module that needs to use the DMA system service must request for channel allocation. An allocated channel is used to setup the channel parameters like the mode of operation(Basic, CRC, chaining etc). Setup the transfer trigger types(Synchronous/Asynchronous). Add a transfer by Setting up the source, destination address and transfer sizes. The DMA transfer starts either forcefully or based on events according to the setting. The channel status events are used to manage the data transfer.

Global Control and Status Management

Provides for control and status of the DMA module. The user can suspend a DMA operation or alternatively resume an already suspended operation. The status of last DMA operation can also be retrieved.

Library Overview

Please refer to the [System Service Introduction](#) for a detailed description of MPLAB Harmony system services.

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

Library Interface Section	Description
Initialization and Task Functions	These functions initialize and enable the DMA subsystem and manage the task State machine.
Channel Setup and Management Functions	These DMA Channel and Setup Management functions enable: <ul style="list-style-type: none"> • Channel Allocation and Release • Setting up of a channel for basic and advanced modes of operations • Channel Enable/Disable • Adding a data transfer • Synchronous data transfer start/abort • Asynchronous data transfer start/abort • Channel Busy status • Channel Computed CRC • Setting the transfer event trigger callback
Global Control and Status Functions	These DMA functions enable global control and status.

How the Library Works

Initialization and Tasks

Describes the functions that can be used for DMA initialization, and provides examples of their usage.

Description

Initialization Function Usage

The DMA subsystem is initialized by calling the initialization routine. The routine also enables the DMA module.

Function Name: SYS_DMA_Initialize

Example

//To Enable the Stop in Idle mode feature

```
SYS_MODULE_OBJ objectHandle;
SYS_DMA_INIT initParam;
```

```
initParam.sidl = SYS_DMA_SIDL_ENABLE;

objectHandle = SYS_DMA_Initialize((SYS_MODULE_INIT*)&initParam); ;
if(SYS_MODULE_OBJ_INVALID == objectHandle)
{
// Handle error
}
```

Task Function Usage

The task routine implements the data transfer state machine and returns a callback on the completion, abortion, or error in a data transfer.

Function Name: SYS_DMA_Tasks

Example

```
// 'objectHandle' Returned from SYS_DMA_Initialize
// Following is the DMA Channel 3 ISR

void __ISR(_DMA3_VECTOR, ipl1AUTO) _IntHandlerSysDmaCh0(void)
{
    SYS_DMA_Tasks(objectHandle, DMA_CHANNEL_3);
}
```



Note: Use SYS_DMA_Tasks(objectHandle) for devices that have a common interrupt vector for all DMA channels.

Function Name: SYS_DMA_TasksError

Example

```
// 'object' Returned from SYS_DMA_Initialize
while (true)
{
    SYS_DMA_TasksError (object);
    // Do other tasks
}
```



Note: The SYS_DMA_TasksError function should be used in Synchronous (polling) mode only. This function should not be called from an ISR.

Function Name: SYS_DMA_TasksISR

Example

```
// 'object' Returned from SYS_DMA_Initialize
// Channel 3 is setup for receiving data by USART peripheral in interrupt mode

// Following is the DMA Channel 3 ISR

void __ISR(_DMA3_VECTOR, ipl5) _InterruptHandler_BT_USART_RX_DMA_CHANNEL(void)
{
    SYS_DMA_TasksISR(object, DMA_CHANNEL_3);
}
```



Note: The SYS_DMA_TasksISR function should be used in Asynchronous (interrupt) mode only. This function should not be called from the SYS_Tasks function.

Function Name: SYS_DMA_TasksErrorISR

Example

```
// 'object' Returned from SYS_DMA_Initialize
// Channel 3 is setup for receiving data by USART peripheral in interrupt mode

// Following is the DMA Channel 3 ISR

void __ISR(_DMA3_VECTOR, ipl5) _InterruptHandler_BT_USART_RX_DMA_CHANNEL(void)
{
    SYS_DMA_TasksErrorISR(object, DMA_CHANNEL_3);
}
```



Note: The SYS_DMA_TasksErrorISR function should be used in Asynchronous (interrupt) mode only. This function should not be called from the SYS_Tasks function.

Channel Setup and Management

Describes the functions that can be used for DMA channel setup and management, and provides examples of their usage.

Description

Channel Allocation and Release Functions

Channel Allocate and Release functions allocate/release a particular channel from the available channels on the particular device.

Channel Allocation Function

Channel Allocate function takes a parameter specifying the requested channel number. If the requested channel is available the function allocates the channel and returns a channel handle. If the user is not particular about any specific channel, the user can specify DMA_CHANNEL_ANY enumerator values. When DMA_CHANNEL_ANY values are specified an available channel is allocated and a channel handle is returned. The function returns an invalid channel handle SYS_DMA_CHANNEL_HANDLE_INVALID when the requested channel is not available for allocation.

The valid channel handle returned by this function must be used in all subsequent DMA channel function calls.

Function Name: SYS_DMA_ChannelAllocate

Example 1

```
/* The following examples requests for allocation of a channel handle with Channel number 2 */
SYS_DMA_CHANNEL_HANDLE handle
DMA_CHANNEL channel;
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
```

Example 2

```
/* The following example requests for allocation of a channel handle with no choice of channel
number */
SYS_DMA_CHANNEL_HANDLE handle
DMA_CHANNEL channel;
channel = DMA_CHANNEL_ANY;
handle = SYS_DMA_ChannelAllocate(channel);
```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Channel Release Function

Channel Release function Dealлокates and frees the DMA channel specified by the channel handle.

Function Name: SYS_DMA_ChannelRelease

Example

```
/* The following example requests for release of a channel handle with Channel number 2 */
DMA_CHANNEL channel;
SYS_DMA_CHANNEL_HANDLE handle;
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelRelease(handle);
```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Channel Setup Functions

The DMA subsystem supports the following modes of operations. The setting up of these modes of operation can be done by the following functions.

- Basic Transfer Mode
- Pattern Match Abort Mode
- Channel Chaining Mode
- Channel Auto Enable Mode
- CRC Mode



The operation modes are not mutually exclusive.

General Channel Setup Function

This function sets up the channel for the supported operating modes. The function does the following:

- Sets up the channel priority
- Enables the specified mode
- Sets up the DMA asynchronous transfer mode

If the DMA channel transfer is intended to be synchronous, the parameter 'eventSrc' (asynchronous trigger source) can be specified as 'DMA_TRIGGER_SOURCE_NONE'. When the channel trigger source is specified as 'DMA_TRIGGER_SOURCE_NONE', The DMA channel transfer needs to be forcefully started by calling the respective function.



- Notes:**
1. Enabling of the available operation mode is not mutually exclusive. More than one operation mode can be enabled by bitwise ORing the operating mode enable parameter.

2. To setup the specific features of the supported operation modes the corresponding function needs to be called after calling this function.

Function Name: SYS_DMA_ChannelSetup

Example

```
/* Configure channel number, priority and enables basic and CRC mode */
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
SYS_DMA_CHANNEL_OP_MODE modeEnable;
DMA_TRIGGER_SOURCE eventSrc;
channel = DMA_CHANNEL_2;
modeEnable = (SYS_DMA_CHANNEL_OP_MODE_BASIC | SYS_DMA_CHANNEL_OP_MODE_CRC);
eventSrc = DMA_TRIGGER_USART_1_TRANSMIT;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelSetup(handle, modeEnable, eventSrc);
```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Pattern Match Abort Mode Function

This function sets up the termination of DMA operation when the specified pattern is matched. Additionally on supported devices, the function also sets up the ignoring of part of a pattern (8-bit) from match abort pattern (16-bit).

Before calling this function the pattern match termination mode must have been enabled by calling the general channel setup function

Function Name: SYS_DMA_ChannelSetupMatchAbortMode

Example 1

```
/* The following code is for a device with 8-bit pattern value and no support for pattern match ignore feature */
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
uint16_t pattern;
DMA_PATTERN_LENGTH length;
SYS_DMA_CHANNEL_IGNORE_MATCH ignore;
uint8_t ignorePattern;
channel = DMA_CHANNEL_2;
priority = DMA_CHANNEL_PRIORITY_1;
pattern = 0x00; //Stop transfer on detection of a NULL character
length = DMA_PATTERN_LENGTH_NONE;
ignore = SYS_DMA_CHANNEL_IGNORE_MATCH_DISABLE;
ignorePattern = 0;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelSetupMatchAbortMode(handle, pattern, length, ignoreEnable, ignorePattern);
```

Example 2

```
/* The following code is for a device with 16-bit pattern value and support for pattern match ignore feature */
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
uint16_t pattern;
DMA_PATTERN_LENGTH length;
```

```

SYS_DMA_CHANNEL_IGNORE_MATCH ignore;
uint8_t                  ignorePattern;
priority      = DMA_CHANNEL_PRIORITY_1;
channel       = DMA_CHANNEL_2;
pattern       = 0x0D0A; //Stop transfer on detection of '\r\n'
length        = DMA_PATTERN_MATCH_LENGTH_2BYTES;
ignore        = SYS_DMA_CHANNEL_IGNORE_MATCH_ENABLE;
ignorePattern = 0x00; \\ Any null character between the termination pattern '\r' and '\n'
is ignored.
handle        = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelSetupMatchAbortMode(handle, pattern, length, ignore, ignorePattern);

```

Preconditions:

1. DMA should be initialized by calling SYS_DMA_Initialize.
2. Channel should be setup and pattern match mode enabled by calling SYS_DMA_ChannelSetup.

CRC Operation Mode Setup Function

This function sets up the CRC computation features of the channel. Before calling this function the CRC mode must have been enabled by calling the general channel setup function

Function Name: SYS_DMA_ChannelSetupCRCMode

Example

```

/* DMA calculation using the CRC background mode */
SYS_DMA_CHANNEL_HANDLE           handle;
DMA_CHANNEL                      channel;
SYS_DMA_CHANNEL_OPERATION_MODE_CRC crc;
channel      = DMA_CHANNEL_2;
crc.type     = DMA_CRC_LFSR;
crc.mode      = SYS_DMA_CHANNEL_CRC_MODE_BACKGROUND;
crc.polyLength = 16;
crc.bitOrder  = DMA_CRC_BIT_ORDER_LSB;
crc.byteOrder = DMA_CRC_BYTEORDER_NO_SWAPPING;
crc.writeOrder = SYS_DMA_CRC_WRITE_ORDER_MAINTAIN;
crc.data      = 0xFFFF;
crc.xorBitMask = 0x1021;
handle       = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelCRCSet(handle, crc);

```

Preconditions:

1. DMA should be initialized by calling SYS_DMA_Initialize.
2. Channel should be setup and CRC mode enabled by calling SYS_DMA_ChannelSetup.

Channel Enable/Disable Functions

The Enable/Disable functions allow to enable/disable a channel on the run.



When a data transfer is added by calling SYS_DMA_ChannelTransferAdd, the channel is automatically enabled.

Note:

Function Name: SYS_DMA_ChannelEnable

Example

```

SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL             channel;
channel      = DMA_CHANNEL_2;
handle       = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelEnable(handle);

```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Function Name: SYS_DMA_ChannelDisable

Example

```

SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL             channel;
channel      = DMA_CHANNEL_2;
handle       = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelDisable(handle);

```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Channel Data Transfer Function

This function adds a data transfer to a DMA channel and Enables the channel to start data transfer. The source and the destination addresses, source and destination lengths, The number of bytes transferred per cell event are set and the channel is enabled to start the data transfer.

Function Name: SYS_DMA_ChannelTransferAdd

Example

```
/* Add 10 bytes of data transfer to UART */
SYS_DMA_CHANNEL_HANDLE handle;
uint8_t buf[10];
void *srcAddr;
void *destAddr;
size_t srcSize;
size_t destSize;
size_t cellSize;
DMA_CHANNEL channel;
channel = DMA_CHANNEL_2;
srcAddr = (uint8_t *) buf;
srcSize = 10;
destAddr = (uint8_t *) &U2TXREG; //Uart 2 TX register is the DMA destination
destSize = 1;
cellSize = 1;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelTransferAdd(handle,srcAddr,srcSize,destAddr,destSize,cellSize);
```

Preconditions:

1. DMA should be initialized by calling SYS_DMA_Initialize.
2. Channel should be setup and enabled by calling SYS_DMA_ChannelSetup.

Synchronous Data Transfer Start/Abort Functions

These functions allows to force start/abort data transfer on the selected channel.



The DMA should have been set up, transfer added before calling these functions.

Function Name: SYS_DMA_ChannelForceStart

Example

```
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelForceStart(handle);
```

Function Name: SYS_DMA_ChannelForceAbort

Example

```
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelForceAbort(handle);
```

Asynchronous Data Transfer Abort Function

This function sets an event source and enables cell transfer abort event for the same for the selected channel.



Note: Asynchronous data transfer is started in general channel setup function. A channel is setup for asynchronous data transfer by default when the appropriate trigger source is specified in general channel setup function call.

Function Name: SYS_DMA_ChannelAbortEventSet

Example

```
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
DMA_TRIGGER_SOURCE eventSrc;
channel = DMA_CHANNEL_2;
```

```
eventSrc = DMA_TRIGGER_CTMU;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelAbortEventSet(handle, eventSrc);
```

Preconditions:

1. DMA should be initialized by calling SYS_DMA_Initialize.
2. Channel should be setup and enabled by calling SYS_DMA_ChannelSetup.

Channel Busy Status Function

This function gets the busy status of the selected channel.

Function Name: SYS_DMA_ChannelsBusy

Example

```
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
bool busyStat;
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
busyStat = SYS_DMA_ChannelIsBusy(handle);
```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Channel Computed CRC Function

This function gets the computed CRC.



Note: The CRC generator must have been previously set up by calling the CRC mode setup function.

To get the computed CRC value this function must be called after the block transfer completion event.

Function Name: SYS_DMA_ChannelGetCRC

Example

```
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
uint32_t computedCRC;
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
computedCRC = SYS_DMA_ChannelCRCSet();
```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Channel Transfer Event Handler Set Function

This function allows to set an event handler for the transfer complete, abort or error events

Function Name: SYS_DMA_ChannelTransferEventHandlerSet

Example

```
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
MY_APP_OBJ myAppObj; //Application specific object
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelTransferEventHandlerSet(handle, APP_DMASYSTransferEventHandler,
(uintptr_t)&myAppObj);

// Event Processing Technique. Event is received when the transfer is processed.

void APP_DMASYSTransferEventHandler(SYS_DMA_TRANSFER_EVENT event, SYS_DMA_CHANNEL_HANDLE
handle, uintptr_t contextHandle)
{
    switch(event)
    {
        case SYS_DMA_TRANSFER_EVENT_COMPLETE:
            // This means the data was transferred.
            break;
        case SYS_DMA_TRANSFER_EVENT_ERROR:
            // Error handling here.
            break;
```

```

    default:
        break;
    }
}

```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Channel Error Get Function

This function returns the error associated with the channel access.

Function Name: SYS_DMA_ChannelErrorGet

Example

```

SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
// Do Channel setup and Transfer Add.
// In the Even Handler Check if there was an Error
if(SYS_DMA_ERROR_ADDRESS_ERROR == SYS_DMA_ChannelErrorGet(handle))
{
    // There was an address error.
    // Do error handling here.
}

```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Global Control and Status

Global Control and Status Function Usage:

Suspend/Resume Functions

These functions allows an enabled DMA module to suspend DMA operations. Operation of the suspended module can also be resumed using these functions.

Function Name: SYS_DMA_Suspend

Example 1

```
SYS_DMA_Suspend();
```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Function Name: SYS_DMA_Resume

Example 2

```
SYS_DMA_Resume();
```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Busy Status Function

This function gets the busy status of DMA module.

Function Name: SYS_DMA_IsBusy

Example

```
bool busyStat;
busyStat = SYS_DMA_IsBusy();
```

Precondition: DMA should be initialized by calling SYS_DMA_Initialize.

Memory to Memory Transfer

Provides a code example for a memory to memory data transfer.

Description

The following code is a typical usage example of using the DMA System Service for a memory to memory data transfer.

```

/* Destination Memory Address pointer */
uint8_t                      *pDmaSrc;
/* Source Memory Address pointer */
uint8_t                      *pDmaDst;

```

```
/* Transfer Size */
size_t txferSize;

/* Maximum Transfer Size */
#define MAX_TRANSFER_SIZE 100
/* The data pattern to be transferred*/
#define SOURCE_TRANSFER_PATTERN 'W'

*****  

The below routine sets up a memory to memory data transfer. It also registers an event handler which is called when the data transfer completes */
void SYS_DMA_TEST_Memory2Memory(void)
{
    uint8_t *pSrcTemp;
    uint32_t index;
    SYS_DMA_INIT dmaInit;
    SYS_DMA_CHANNEL_HANDLE channelHandle;

    /* Initializing source and destination variables */
    pDmaSrc = (uint8_t *) NULL;
    pDmaDst = (uint8_t *) NULL;
    pSrcTemp = (uint8_t *) NULL;
    txferSize = MAX_TRANSFER_SIZE;
    pDmaSrc = (uint8_t *) malloc(txferSize);
    pDmaDst = (uint8_t *) malloc(txferSize);
    pSrcTemp = pDmaSrc;

    if((uint8_t *) NULL != pDmaSrc) && ((uint8_t *) NULL != pDmaDst))
    {
        /* Initialize the source memory block with the pattern */
        for(index=0; index < txferSize; index++)
        {
            *pSrcTemp++= SOURCE_TRANSFER_PATTERN;
        }

        /* Initialize the DMA system service */
        dmaInit.sidl = SYS_DMA_SIDL_DISABLE;
        sysObj = SYS_DMA_Initialize((SYS_MODULE_INIT*)&dmaInit);

        /* Allocate a DMA channel */
        channelHandle = SYS_DMA_ChannelAllocate(DMA_CHANNEL_1);
        if(SYS_DMA_CHANNEL_HANDLE_INVALID != channelHandle)
        {
            /* Register an event handler for the channel */
            SYS_DMA_ChannelTransferEventHandlerSet(channelHandle,
                Sys_DMA_Mem2Mem_Event_Handler, (uintptr_t)&sysContext);

            /* Setup the channel */
            SYS_DMA_ChannelSetup(channelHandle,
                SYS_DMA_CHANNEL_OP_MODE_BASIC,
                DMA_TRIGGER_SOURCE_NONE);

            /* Add the memory block transfer request */
            SYS_DMA_ChannelTransferAdd(channelHandle, pDmaSrc, txferSize,
                pDmaDst, txferSize, txferSize);

            /* Start the DMA transfer */
            SYS_DMA_ChannelForceStart(channelHandle);
        }
        else
        {
            /* Channel Handle not available */
            ;
        }
    }
}
```

```

        }

/* Below is an Event handler for the registered memory to memory DMA channel */
static void Sys_DMA_Mem2Mem_Event_Handler(SYS_DMA_TRANSFER_EVENT event,
    SYS_DMA_CHANNEL_HANDLE handle, uintptr_t contextHandle)
{
    int32_t fail=0;
    uint32_t index;

    /* Success event */
    if(SYS_DMA_TRANSFER_EVENT_COMPLETE == event)
    {
        /* Verify the contents of destination block matches the
         * memory contents of the source block */
        for(index=0; index < txferSize; index++)
        {
            if(*pDmaSrc++!=*pDmaDst++) // compare the buffers
            {
                fail = -1;
            }
        }
        if(0==fail)
        {
            /* Transfer Success */
        }
        else
        {
            /* Transfer Failed */
        }
    }
    /* Failure Event */
    else if(SYS_DMA_TRANSFER_EVENT_ABORT == event)
    {
    }
}

```

Configuring the Library

The configuration of the DMA System Service is based on the file `system_config.h`

This header file contains the configuration selection for the DMA system service. Based on the selections made, the DMA System Service may support the selected features. These configuration settings will apply to all instances of the DMA System Service.

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 Help section for more details.

Building the Library

This section lists the files that are available in the Direct Memory Access (DMA) System Service 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/system/dma`.

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
<code>sys_dma.h</code>	DMA System Service Library API header file.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

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/sys_dma.c	DMA System Service Library implementation.

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 for the DMA System Service Library.

Module Dependencies

The DMA System Service Library does not depend on any other modules.

Library Interface

This section describes the APIs of the DMA System Service Library.

Refer to each section for a detailed description.

Data Types and Constants

Files

This section lists the source and header files used by the library.

File System Service Library Help

This section describes the File System Service Library.

Introduction

Introduction to the MPLAB Harmony File System (FS).

Description

The MPLAB Harmony File System (FS) provides file system services to MPLAB Harmony based applications. The architecture of the File System Service is shown in the following figure.

The File System Service provides an application programming interface (API) through which a utility or user program requests services of a file system. Some file system APIs may also include interfaces for maintenance operations, such as creating or initializing a file system and verifying the file system for integrity.

The File System service is really a framework designed to support multiple file systems (native file system) and multiple media in the same application. Examples of native file systems are FAT12, FAT16, FAT32, and the Microchip File System (MPFS) among others.

MPFS is a read-only file system, which is used to store the files that do not change for example Web pages, images, etc. Each of these native file systems have a common set of APIs that can be used to access the files of that particular native file system.

The File System Service abstracts the native file system calls and provides a common interface to the user/application layer. For example, while the application layer requests for a file read or write from a disk, due to the presence of the this abstraction, the application need not be bothered about the native file system implemented on that disk. Instead, the application can call the read/write API of the File System, which in turn translates to the read/write command of the native file system used on the required disk.

This simplifies the implementation of the higher application layer and also provides a mechanism to add more native file system to the File System framework in the future.

 **Note:** "File System Service" and "sys_fs" are synonymous.

Using the Library

This topic describes the basic architecture of the File System Service Library and provides information on how to use it.

Description

Interface Header File: `sys_fs.h`

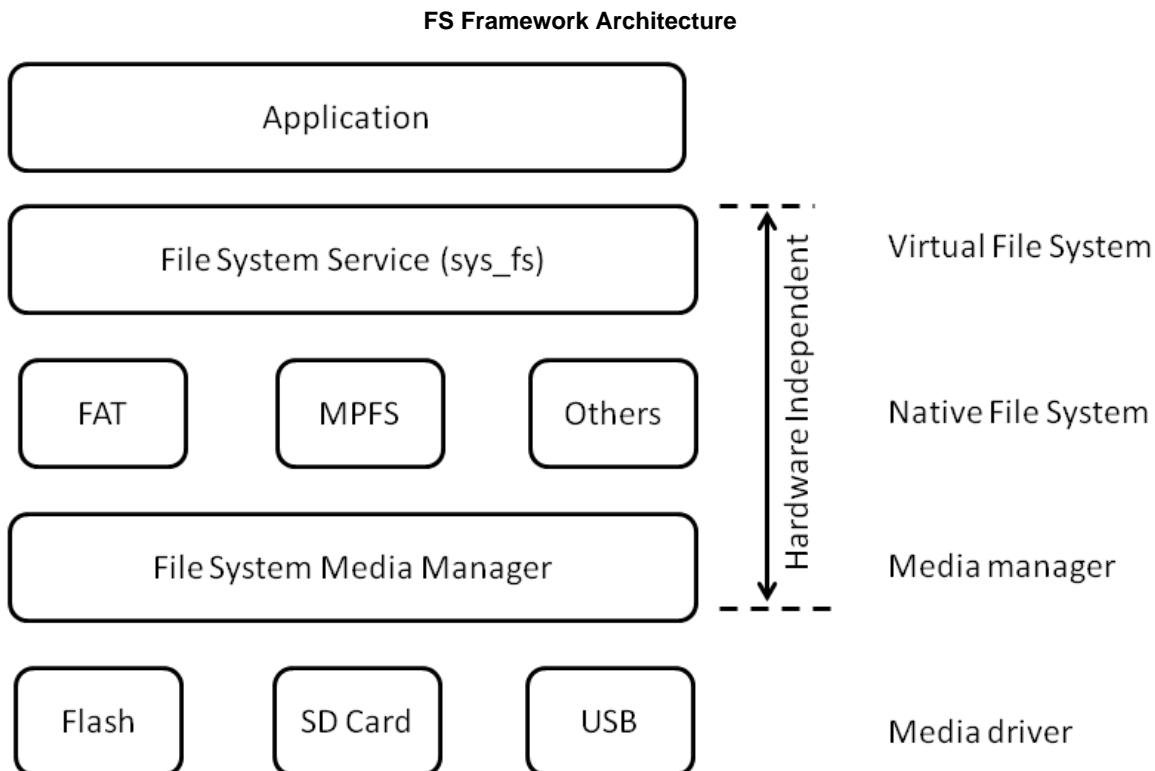
The interface to the File System Service library is defined in the `sys_fs.h` header file. Any C language source (.c) file that uses the File System Service Library use this API's to perform file operation on various connected media's.

Abstraction Model

This topic describes the abstraction model of the MPLAB Harmony File System.

Description

The FS framework features a modular and layered architecture, as shown in the following figure.



As seen in the figure, the FS Framework consists of the following major blocks:

- The **Driver** for the physical media has to be included as a part of the FS Framework. This layer provides a low-level interface to access the physical media. This layer also enables multiple instances of media. Examples of drivers are:
 - Memory driver – To access files using Various Flash Memories (QSPI Flash, NVM Flash, EEPROM Flash)
 - SDCARD driver – To access files from SD card
- The **Media driver** provides a mechanism to access the media as "sectors". Sectors are the smallest storage element accessed by a file system and are contiguous memory locations. Typically, each sector has 512 bytes. Depending on the requirement, in some cases, the driver and media driver could be combined as one layer.
- The **Media manager** implements a disk and sector based media access mechanism. It also performs disk allocated/deallocated on media attach/detach. Due to the implementation of this layer, the FS Framework can support multiple disks. The media manager detects and analyzes a media based on its Master Boot Record (MBR). Therefore, it is mandatory for the media to have a MBR for it to work with the FS.
- The **Native file system** implements support for the media file system format. Examples of native file systems are: FAT12, FAT32, and MPFS, among other. At present, only the FAT and MPFS file systems are supported by the FS framework; however, more native file systems can be included.
- The **Virtual file system (or SYS_FS)** layer provides a file system independent file system operation interface. This layer translates virtual file systems calls to native file system calls. Due to this layer, applications can now support multiple file systems. Interfaces provided by this layer, but not limited to, include:
 - SYS_FS_mount
 - SYS_FS_open
 - SYS_FS_read
 - SYS_FS_write
 - SYS_FS_close

How the Library Works

This topic provides information on how the MPLAB Harmony File System works.

Description

The MPLAB Harmony File System (FS) provides embedded application developers with a file system framework for retrieving and storing data from various media.

The MPLAB Harmony file system is designed to support multiple file systems (native file systems) and multiple media at the same time. Examples of native file systems are FAT12, FAT32, MPFS, and JFS, among others. Each of these native file systems has a common set of APIs that can be used to access the files of that particular native file system. The FS is a part of the MPLAB Harmony installation and is accompanied by demonstration applications that highlight usage. These demonstrations can also be modified or updated to build custom applications.

FS features include the following:

- Support for multiple file system (FAT, MPFS)
- Supports multiple physical media (NVM, SPI_FLASH, SD card)
- More physical media can be interfaced with the FS, once the driver is available for the media
- Modular and Layered architecture

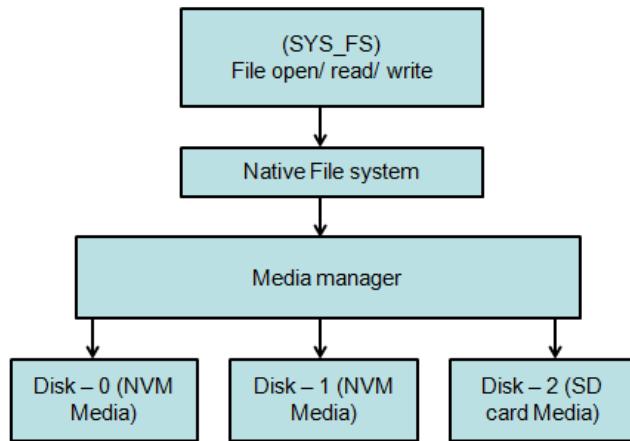
Application Interaction

This topic describes how an application must interact with the File System.

Description

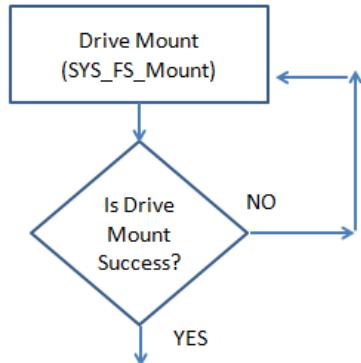
The interaction of various layers is shown in the following figure.

Application Interaction with FS Framework



In the process of using the FS Framework, the application must first **mount** the media drive for the FS Framework to access the media. Unless the mounting process returns successfully, the application should continue trying to mount the drive. If the drive is not attached, the mounting process will fail. In such a situation, the application should not proceed further unless the mounting is success.

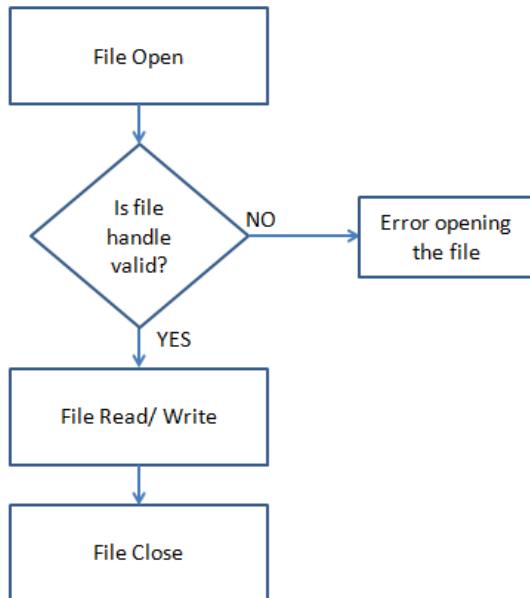
Application Mounts a Drive



Once the drive is mounted, the application code can then **open the file** from the drive with different attributes (such as read-only or write). If the file open returns a valid handle, the application can proceed further. Otherwise, the application will enter an error state. The reason for an invalid handle could be that the application was trying to read a file from the drive that does not exist. Another reason for an invalid handle is when the application tries to write to a drive that is write-protected.

Once the file is opened, the valid file handle is further used to **read/write** data to the file. Once the required operation is performed on the file, the file can then be **closed** by the application by passing the file handle. The following figure illustrates the process.

Further File System Operations



Using the File System

This topic describes how to use the File System.

Description

Use the Available Library Demonstration Applications

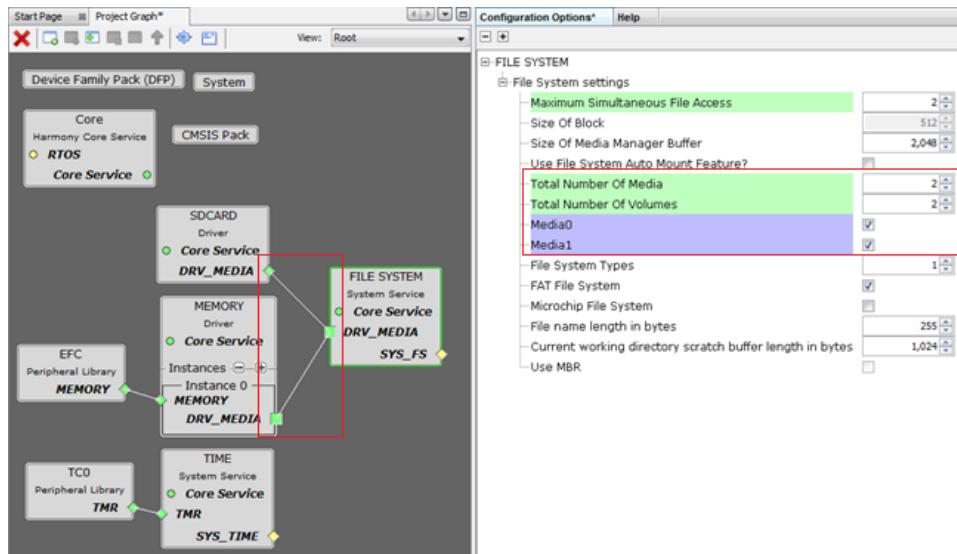
The FS framework release package contains a set of demonstration applications that are representative of common scenario (single/multi-media and single/multi-native file systems). These demonstrations can be easily modified to include application-specific initialization and application logic. The application logic must be non-blocking and could be implemented as a state machine.

Configuring the Library

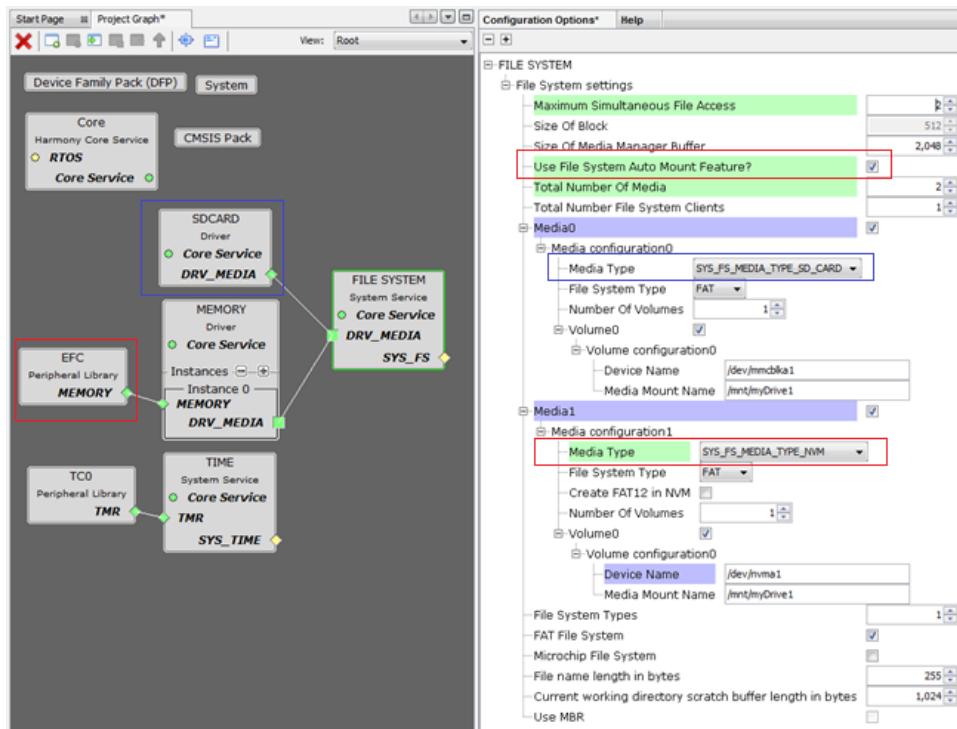
This Section provides information on how to configure File System Service Library.

Description

File System Service Library should be configured via MHC. Below are the Snapshot of the MHC configuration window for Memory driver and brief description.



File System Configuration with Auto Mount Disabled



File System Configuration with Auto Mount Enabled

Configuration Options:

1. Maximum Simultaneous File Access

1. Maximum Number of files which can be accessed by application

2. Size Of Block

1. Block Size used by File System layer to chunk the application data and to send to attached media

3. Size Of Media Manager Buffer

1. Media Manager Buffer size to store Block Data

4. Use File System Auto Mount Feature

1. Enables auto mount feature for all the media's

5. Total Number Of Media

1. Number of media to be attached to file system

6. Total Number Of Volumes

1. Number of volumes to be created for each media

7. MediaX

1. Media details to be configured when Auto mount feature is enabled.

8. File System Types

1. Number of file systems used

9. FAT File System

1. When Selected Specifies FAT File system will be used

10. Microchip File System

1. When Selected Specifies MPFS File system will be used

11. File Name Length

1. Max file name length to be supported

12. Current Working Directory Scratch buffer length in Bytes

1. Buffer size to store the current directory path

13. Use MBR

1. Use Master Boot Record.

Building the Library

This section lists the files that are available in the File System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library.

Interface File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

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
/sys_fs.h	This file contains function and type declarations required to interact with the MPLAB Harmony File System Framework.
/fat_fs/src/file_system/ff.h	FAT File System module include file. This file should be included when using the FAT File System.
/mpfs/mpfs.h	This file contains the interface definition for handling the Microchip File System (MPFS). This file should be included when using MPFS.

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/sys_fs.c	This file contains implementation of File System interfaces.
/src/dynamic/sys_fs_media_manager.c	This file contains implementation File System Media Manager functions.
/fat_fs/src/file_system/ff.c	This file implements the FAT File system functions. This file should be included when using FAT File System.
/fat_fs/src/hardware_access/diskio.c	Low-level disk I/O module for FAT File System. This file should be included when using FAT File System.
/mpfs/src/mpfs.c	This file implements the MPFS functions. This file should be included when using MPFS.

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 for this library.

Module Dependencies

The File System Service Library depends on the following modules:

- [Memory](#) Driver Library (if media is NVM, SPI flash, EEPROM Flash)
- Secure Digital (SD) Card Driver Library (if media is a SD Card)

Library Interface

a) File and Directory Operation Functions

	Name	Description
≡◊	SYS_FS_FileOpen	Opens a file.
≡◊	SYS_FS_FileClose	Closes a file.
≡◊	SYS_FS_FileCharacterPut	Writes a character to a file.
≡◊	SYS_FS_FileEOF	Checks for end of file.
≡◊	SYS_FS_FileError	Returns the file specific error.
≡◊	SYS_FS_FileNameGet	Reads the file name.
≡◊	SYS_FS_FilePrintf	Writes a formatted string into a file.
≡◊	SYS_FS_FileRead	Read data from the file.
≡◊	SYS_FS_FileSeek	Moves the file pointer by the requested offset.
≡◊	SYS_FS_FileSize	Returns the size of the file in bytes.
≡◊	SYS_FS_FileStat	Gets file status.
≡◊	SYS_FS_FileStringGet	Reads a string from the file into a buffer.
≡◊	SYS_FS_FileStringPut	Writes a string to a file.
≡◊	SYS_FSFileSync	Flushes the cached information when writing to a file.
≡◊	SYS_FS_FileTell	Obtains the file pointer position.
≡◊	SYS_FS_FileTestError	Checks for errors in the file.
≡◊	SYS_FS_FileTruncate	Truncates a file
≡◊	SYS_FS_FileWrite	Writes data to the file.
≡◊	SYS_FS_DirectoryMake	Makes a directory.
≡◊	SYS_FS_DirOpen	Open a directory
≡◊	SYS_FS_DirClose	Closes an opened directory.
≡◊	SYS_FS_DirRead	Reads the files and directories of the specified directory.
≡◊	SYS_FS_DirRewind	Rewinds to the beginning of the directory.
≡◊	SYS_FS_DirSearch	Searches for a file or directory.
≡◊	SYS_FS_DirectoryChange	Changes to a the directory specified.
≡◊	SYS_FS_CurrentWorkingDirectoryGet	Gets the current working directory
≡◊	SYS_FS_FileDirectoryModeSet	Sets the mode for the file or directory.
≡◊	SYS_FS_FileDirectoryRemove	Removes a file or directory.
≡◊	SYS_FS_FileDirectoryRenameMove	Renames or moves a file or directory.
≡◊	SYS_FS_FileDirectoryTimeSet	Sets or changes the time for a file or directory.
≡◊	SYS_FS_Error	Returns the last error.

b) General Operation Functions

	Name	Description
≡	SYS_FS_Initialize	Initializes the file system abstraction layer (sys_fs layer).
≡	SYS_FS_Tasks	Maintains the File System tasks and functionalities.
≡	SYS_FS_Mount	Mounts the file system.
≡	SYS_FS_Unmount	Unmounts the file system.
≡	SYS_FS_CurrentDriveGet	Gets the current drive
≡	SYS_FS_CurrentDriveSet	Sets the drive.
≡	SYS_FS_DriveLabelGet	Gets the drive label.
≡	SYS_FS_DriveLabelSet	Sets the drive label
≡	SYS_FS_DriveFormat	Formats a drive.
≡	SYS_FS_DrivePartition	Partitions a physical drive (media).
≡	SYS_FS_DriveSectorGet	Obtains total number of sectors and number of free sectors for the specified drive.
≡	SYS_FS_EventHandlerSet	Allows a client to identify an event handling function for the file system to call back when mount/unmount operation has completed.

c) Media Manager Functions

	Name	Description
≡	SYS_FS_MEDIA_MANAGER_Tasks	Media manager task function.
≡	SYS_FS_MEDIA_MANAGER_TransferTask	Media manager transfer task function.
≡	SYS_FS_MEDIA_MANAGER_Read	Gets data from a specific media address.
≡	SYS_FS_MEDIA_MANAGER_SectorRead	Reads a specified media sector.
≡	SYS_FS_MEDIA_MANAGER_SectorWrite	Writes a sector to the specified media.
≡	SYS_FS_MEDIA_MANAGER_Register	Function to register media drivers with the media manager.
≡	SYS_FS_MEDIA_MANAGER_RegisterTransferHandler	Register the event handler for data transfer events.
≡	SYS_FS_MEDIA_MANAGER_DeRegister	Function called by a media to deregister itself to the media manager. For static media, (like NVM or SD card), this "deregister function" is never called, since static media never gets deregistered once they are initialized. For dynamic media (like MSD), this register function is called dynamically, once the MSD media is connected.
≡	SYS_FS_MEDIA_MANAGER_AddressGet	Gets the starting media address based on a disk number.
≡	SYS_FS_MEDIA_MANAGER_MediaStatusGet	Gets the media status.
≡	SYS_FS_MEDIA_MANAGER_VolumePropertyGet	Gets the volume property.
≡	SYS_FS_MEDIA_MANAGER_CommandStatusGet	Gets the command status.
≡	SYS_FS_MEDIA_MANAGER_GetMediaGeometry	Gets the media geometry information.
≡	SYS_FS_MEDIA_MANAGER_EventHandlerSet	Register the event handler for Mount/Un-Mount events.

d) File System Data Types and Constants

	Name	Description
	SYS_FS_ERROR	Lists the various error cases.
	SYS_FS_FILE_SEEK_CONTROL	Lists the various modes of file seek.
	SYS_FS_FSTAT	File System status
	SYS_FS_FUNCTIONS	SYS FS Function signature structure for native file systems.
	SYS_FS_REGISTRATION_TABLE	The sys_fs layer has to be initialized by passing this structure with suitably initialized members.
	SYS_FS_RESULT	Lists the various results of a file operation.
	SYS_FS_FILE_OPEN_ATTRIBUTES	Lists the various attributes (modes) in which a file can be opened.
	FAT_FS_MAX_LFN	Maximum length of the Long File Name.
	FAT_FS_MAX_SS	Lists the definitions for FAT file system sector size.

	FAT_FS_USE_LFN	Lists the definitions for FAT file system LFN selection.
	SYS_FS_FILE_SYSTEM_TYPE	Enumerated data type identifying native file systems supported.
	SYS_FS_HANDLE	This type defines the file handle.
	SYS_FS_HANDLE_INVALID	Invalid file handle
	SYS_FS_FILE_DIR_ATTR	Enumerated data type identifying the various attributes for file/directory.
	SYS_FS_TIME	The structure to specify the time for a file or directory.
	SYS_FS_FORMAT	Specifies the partitioning rule.
	SYS_FS_EVENT	Identifies the possible file system events.
	SYS_FS_EVENT_HANDLER	Pointer to the File system Handler function.
	SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID	Defines the invalid media block command handle.
	SYS_FS_MEDIA_HANDLE_INVALID	Defines the invalid media handle.
	_SYS_FS_VOLUME_PROPERTY	Structure to obtain the property of volume
	SYS_FS_MEDIA_BLOCK_EVENT	Identifies the possible events that can result from a request.
	SYS_FS_MEDIA_COMMAND_STATUS	The enumeration for status of buffer
	SYS_FS_MEDIA_FUNCTIONS	Structure of function pointers for media driver
	SYS_FS_MEDIA_MOUNT_DATA	Structure to obtain the device and mount name of media
	SYS_FS_MEDIA_PROPERTY	Contains information of property of a media.
	SYS_FS_MEDIA_STATE	The enumeration for state of media.
	SYS_FS_MEDIA_STATUS	The state of media.
	SYS_FS_MEDIA_TYPE	The enumeration for type of media.
	SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the media.
	SYS_FS_MEDIA_EVENT_HANDLER	Pointer to the Media Event Handler function.
	SYS_FS_MEDIA_GEOMETRY	Contains all the geometrical information of a media device.
	SYS_FS_MEDIA_HANDLE	Handle identifying the media registered with the media manager.
	SYS_FS_MEDIA_REGION_GEOMETRY	Contains information of a sys media region.
	SYS_FS_VOLUME_PROPERTY	Structure to obtain the property of volume

Description

This section describes the APIs of the File System Service Library.

Refer to each section for a detailed description.

a) File and Directory Operation Functions

SYS_FS_FileOpen Function

Opens a file.

File

[sys_fs.h](#)

C

```
SYS_FS_HANDLE SYS_FS_FileOpen(const char* fname, SYS_FS_FILE_OPEN_ATTRIBUTES attributes);
```

Returns

On success - A valid file handle will be returned On failure - [SYS_FS_HANDLE_INVALID](#). The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function opens a file with the requested attributes.

Remarks

None.

Preconditions

Prior to opening a file, the name of the volume on which the file resides should be known and the volume should be mounted.

Example

```

SYS_FS_HANDLE fileHandle;

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG" ,
    (SYS_FS_FILE_OPEN_READ) );

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open succeeded.
}
else
{
    // File open failed.
}

// Using SYS_FS_CurrentDriveSet () function.

SYS_FS_HANDLE fileHandle;

SYS_FS_CurrentDriveSet( "/mnt/myDrive" );

fileHandle = SYS_FS_FileOpen( "FILE.JPG" , (SYS_FS_FILE_OPEN_READ) );
if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open succeeded.
}
else
{
    // File open failed.
}

```

Parameters

Parameters	Description
fname	The name of the file to be opened along with the path. The fname format is as follows
attributes	"/mnt/volumeName/dirName/fileName". volumeName is the name of the volume/drive. dirName is the name of the directory under which the file is located. fileName is the name of the file to be opened. The "/mnt/volumeName" portion from the fName can be omitted if the SYS_FS_CurrentDriveSet () has been invoked to set the current drive/volume.
	Access mode of the file, of type SYS_FS_FILE_OPEN_ATTRIBUTES

Function

```

SYS_FS_HANDLE SYS_FS_FileOpen
(
    const char* fname,
    SYS_FS_FILE_OPEN_ATTRIBUTES attributes
);

```

SYS_FS_FileClose Function

Closes a file.

File

`sys_fs.h`

C

```
SYS_FS_RESULT SYS_FS_FileClose(SYS_FS_HANDLE handle);
```

Returns

`SYS_FS_RES_SUCCESS` - File close operation was successful. `SYS_FS_RES_FAILURE` - File close operation failed. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function closes an opened file.

Remarks

None.

Preconditions

A valid file handle must be obtained before closing a file.

Example

```
SYS_FS_HANDLE fileHandle;
fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG" ,
    (SYS_FS_FILE_OPEN_READ)) ;

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

SYS_FS_FileClose(fileHandle);
```

Parameters

Parameters	Description
<code>handle</code>	A valid handle, which was obtained while opening the file.

Function

```
SYS_FS_RESULT SYS_FS_FileClose
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileCharacterPut Function

Writes a character to a file.

File

`sys_fs.h`

C

```
SYS_FS_RESULT SYS_FS_FileCharacterPut(SYS_FS_HANDLE handle, char data);
```

Returns

SYS_FS_RES_SUCCESS - Write operation was successful. SYS_FS_RES_FAILURE - Write operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function writes a character to a file.

Remarks

None.

Preconditions

The file into which a character has to be written, has to be present and should have been opened.

Example

```
SYS_FS_RESULT res;
SYS_FS_HANDLE fileHandle;

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG", (SYS_FS_FILE_OPEN_WRITE_PLUS) );
if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

// Write a character to the file.
res = SYS_FS_FileCharacterPut(fileHandle, 'c');
if(res != SYS_FS_RES_SUCCESS)
{
    // Character write operation failed.
}
```

Parameters

Parameters	Description
handle	file handle to which the character is to be written.
data	character to be written to the file.

Function

```
SYS_FS_RESULT SYS_FS_FileCharacterPut
(
    SYS_FS_HANDLE handle,
    char data
);
```

SYS_FS_FileEOF Function

Checks for end of file.

File

[sys_fs.h](#)

C

```
bool SYS_FS_FileEOF(SYS_FS_HANDLE handle);
```

Returns

On success returns true indicating that the file pointer has reached the end of the file. On failure returns false. This could be due to file pointer having not reached the end of the file. Or due to an invalid file handle. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

Checks whether or not the file position indicator is at the end of the file.

Remarks

None.

Preconditions

A valid file handle must be obtained before knowing a EOF.

Example

```
SYS_FS_HANDLE fileHandle;
bool eof;

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG",
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}
...
...

eof = SYS_FS_FileEOF(fileHandle);

if(eof == false)
{
    // Check the error state using SYS_FS_FileError
}
```

Parameters

Parameters	Description
handle	file handle obtained during file Open.

Function

```
bool SYS_FS_FileEOF
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileError Function

Returns the file specific error.

File

`sys_fs.h`

C

```
SYS_FS_ERROR SYS_FS_FileError(SYS_FS_HANDLE handle);
```

Returns

Error code of type `SYS_FS_ERROR`.

Description

For file system functions which accepts valid handle, any error happening in those functions could be retrieved with `SYS_FS_FileError`. This function returns errors which are file specific.

Please note that if an invalid handle is passed to a file system function, in such a case, `SYS_FS_FileError` will not return the

correct type of error, as the handle was invalid. Therefore, it would be prudent to check the errors using the [SYS_FS_Error](#) function.

Remarks

None.

Preconditions

This function has to be called immediately after a failure is observed while doing a file operation. Any subsequent failure will overwrite the cause of previous failure.

Example

```
...
const char *buf = "Hello World";
size_t nbytes;
size_t bytes_written;
SYS_FS_HANDLE fd;
SYS_FS_ERROR err;
...

bytes_written = SYS_FS_FileWrite((const void *)buf, nbytes, fd);

if(bytes_written == -1)
{
    // error while writing file
    // find the type (reason) of error
    err = SYS_FS_FileError(fd);
}
```

Parameters

Parameters	Description
handle	A valid file handle

Function

```
SYS_FS_ERROR SYS_FS_FileError
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileNameGet Function

Reads the file name.

File

[sys_fs.h](#)

C

```
bool SYS_FS_FileNameGet(SYS_FS_HANDLE handle, uint8_t* cName, uint16_t wLen);
```

Returns

Returns true if the file name was read successfully. Returns false if the file name was not read successfully. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function reads the file name of a file that is already open.

Remarks

None.

Preconditions

The file handle referenced by handle is already open.

Example

```

SYS_FS_HANDLE fileHandle;
bool stat;
uint8_t fileName[255];

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG",
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}
...
...

stat = SYS_FS_FileNameGet(fileHandle, fileName, 8 );

if(stat == false)
{
    // file not located based on handle passed
    // Check the error state using SYS_FS_FileError
}

```

Parameters

Parameters	Description
handle	File handle obtained during file Open.
cName	Where to store the name of the file.
wLen	The maximum length of data to store in cName.

Function

```

bool SYS_FS_FileNameGet
(
    SYS_FS_HANDLE handle,
    uint8_t* cName,
    uint16_t wLen
);

```

SYS_FS_FilePrintf Function

Writes a formatted string into a file.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FilePrintf(SYS_FS_HANDLE handle, const char * string, ...);
```

Returns

SYS_FS_RES_SUCCESS - Formatted string write operation was successful. SYS_FS_RES_FAILURE - Formatted string write operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function writes a formatted string into a file.

Remarks

None.

Preconditions

The file into which a string has to be written, must exist and should be open.

Example

```

SYS_FS_RESULT res;
SYS_FS_HANDLE fileHandle;

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.txt", (SYS_FS_FILE_OPEN_WRITE_PLUS) );

if( fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

// Write a string
res = SYS_FS_FilePrintf(fileHandle, "%d", 1234);
if( res != SYS_FS_RES_SUCCESS)
{
    // write operation failed.
}

```

Parameters

Parameters	Description
handle	File handle to which formatted string is to be written.
string	Pointer to formatted string which has to be written into file.

Function

```

SYS_FS_RESULT SYS_FS_FilePrintf
(
    SYS_FS_HANDLE handle,
    const char *string,
    ...
);

```

SYS_FS_FileRead Function

Read data from the file.

File

[sys_fs.h](#)

C

```
size_t SYS_FS_FileRead(SYS_FS_HANDLE handle, void * buf, size_t nbyte);
```

Returns

On success returns the number of bytes read successfully(0 or positive number). On failure returns -1. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function attempts to read nbyte bytes of data from the file associated with the file handle into the buffer pointed to by buf.

Remarks

None.

Preconditions

A valid file handle must be obtained before reading a file.

Example

```
...
char buf[20];
size_t nbytes;
size_t bytes_read;
SYS_FS_HANDLE fd;
...
nbytes = sizeof(buf);
bytes_read = SYS_FS_FileRead(fd, buf, nbytes);
...
```

Parameters

Parameters	Description
handle	File handle obtained during file open.
buf	Pointer to buffer into which data is read.
nbyte	Number of bytes to be read

Function

```
size_t SYS_FS_FileRead
(
    SYS_FS_HANDLE handle,
    void *buf,
    size_t nbyte
);
```

SYS_FS_FileSeek Function

Moves the file pointer by the requested offset.

File

[sys_fs.h](#)

C

```
int32_t SYS_FS_FileSeek(SYS_FS_HANDLE fildes, int32_t offset, SYS_FS_FILE_SEEK_CONTROL whence);
```

Returns

On success - The number of bytes by which file pointer is moved (0 or positive number) On Failure - (-1) If the chosen offset value was (-1), the success or failure can be determined with [SYS_FS_Error](#).

Description

This function sets the file pointer for a open file associated with the file handle, as follows: whence = SYS_FS_SEEK_SET - File offset is set to offset bytes from the beginning. whence = SYS_FS_SEEK_CUR - File offset is set to its current location plus offset. whence = SYS_FS_SEEK_END - File offset is set to the size of the file plus offset. The offset specified for this option should be negative for the file pointer to be valid.

Trying to move the file pointer using SYS_FS_FileSeek, beyond the range of file will only cause the pointer to be moved to the last location of the file.

Remarks

None.

Preconditions

A valid file handle must be obtained before seeking a file.

Example

```

SYS_FS_HANDLE fileHandle;
int status;

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG",
                             (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}
...
...

status = SYS_FS_FileSeek(fileHandle, 5, SYS_FS_SEEK_CUR);

if((status != -1) && (status == 5))
{
    // Success
}

```

Parameters

Parameters	Description
handle	A valid file handle obtained during file open.
offset	The number of bytes which act as file offset. This value could be a positive or negative value.
whence	Type of File Seek operation as specified in SYS_FS_FILE_SEEK_CONTROL .

Function

```

int32_t SYS_FS_FileSeek
(
    SYS_FS_HANDLE handle,
    int32_t offset,
    SYS_FS_FILE_SEEK_CONTROL whence
);

```

SYS_FS_FileSize Function

Returns the size of the file in bytes.

File

[sys_fs.h](#)

C

```
int32_t SYS_FS_FileSize(SYS_FS_HANDLE handle);
```

Returns

On success returns the size of the file in bytes. On failure returns -1. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function returns the size of the file as pointed by the handle.

Remarks

None.

Preconditions

A valid file handle must be obtained before knowing a file size.

Example

```

SYS_FS_HANDLE fileHandle;
long fileSize;

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG" ,
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}
...
...

fileSize = SYS_FS_FileSize(fileHandle);

if(fileSize != -1)
{
    // Success
}

```

Parameters

Parameters	Description
handle	File handle obtained during file Open.

Function

```

int32_t SYS_FS_FileSize
(
    SYS_FS_HANDLE handle
);

```

SYS_FS_FileStat Function

Gets file status.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileStat(const char * fname, SYS_FS_FSTAT * buf);
```

Returns

SYS_FS_RES_SUCCESS - File stat operation was successful. SYS_FS_RES_FAILURE - File stat operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function obtains information about a file associated with the file name, and populates the information in the structure pointed to by buf. This function can read the status of file regardless of whether a file is opened or not.

Remarks

None.

Preconditions

Prior to opening a file, the name of the volume on which the file resides should be known and the volume should be mounted.

Example

```
SYS_FS_fStat fileStat;

if(SYS_FS_FileStat("/mnt/myDrive/FILE.TXT", &fileStat) == SYS_FS_RES_SUCCESS)
{
    // Successfully read the status of file "FILE.TXT"
}
```

Parameters

Parameters	Description
fname	Name of the file with the path and the volume name. The string of volume and file name has to be preceded by "/mnt/". Also, the volume name and file name has to be separated by a slash "/".
buf	pointer to SYS_FS_FSTAT structure.

Function

```
SYS_FS_RESULT SYS_FS_FileStat
(
const char *fname,
SYS_FS_FSTAT *buf
)
```

SYS_FS_FileStringGet Function

Reads a string from the file into a buffer.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileStringGet(SYS_FS_HANDLE handle, char* buff, uint32_t len);
```

Returns

SYS_FS_RES_SUCCESS - String read operation was successful. SYS_FS_RES_FAILURE - String read operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function reads a string of specified length from the file into a buffer. The read operation continues until

1. 'n' is stored
2. reached end of the file or
3. the buffer is filled with len - 1 characters.

The read string is terminated with a '0'.

Remarks

None.

Preconditions

The file from which a string has to be read, has to be present and should have been opened.

Example

```
SYS_FS_RESULT res;
SYS_FS_HANDLE fileHandle;
char buffer[100];

fileHandle = SYS_FS_FileOpen ("/mnt/myDrive/FILE.JPG", (SYS_FS_FILE_OPEN_WRITE_PLUS));
if(fileHandle != SYS_FS_HANDLE_INVALID)
```

```

{
    // File open is successful
}

// Read a string from the file.
res = SYS_FS_FileStringGet(fileHandle, buffer, 50);
if( res != SYS_FS_RES_SUCCESS)
{
    //String read operation failed.
}

```

Parameters

Parameters	Description
handle	Handle of the file from which string is to be read.
buff	Buffer in which the string is to be stored.
len	length of string to be read.

Function

```

SYS_FS_RESULT SYS_FS_FileStringGet
(
    SYS_FS_HANDLE handle,
    char* buff,
    uint32_t len
);

```

SYS_FS_FileStringPut Function

Writes a string to a file.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileStringPut(SYS_FS_HANDLE handle, const char * string);
```

Returns

SYS_FS_RES_SUCCESS - String write operation was successful. SYS_FS_RES_FAILURE - String write operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function writes a string into a file. The string to be written should be NULL terminated. The terminator character will not be written.

Remarks

None.

Preconditions

The file into which a string has to be written, has to be present and should have been opened.

Example

```

SYS_FS_RESULT res;
SYS_FS_HANDLE fileHandle;

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG", SYS_FS_FILE_OPEN_WRITE_PLUS );
if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

```

```

// Write a string
res = SYS_FS_FileStringPut(fileHandle, "Hello World");
if(res != SYS_FS_RES_SUCCESS)
{
    // String write operation failed.
}

```

Parameters

Parameters	Description
handle	File handle to which string is to be written.
string	Pointer to the null terminated string which has to be written into file.

Function

```

SYS_FS_RESULT SYS_FS_FileStringPut
(
    SYS_FS_HANDLE handle,
    const char *string
);

```

SYS_FSFileSync Function

Flushes the cached information when writing to a file.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FSFileSync(SYS_FS_HANDLE handle);
```

Returns

SYS_FS_RES_SUCCESS - File sync operation was successful. SYS_FS_RES_FAILURE - File sync operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function flushes the cached information when writing to a file. The SYS_FSFileSync function performs the same process as [SYS_FS_FileClose](#) function; however, the file is left open and can continue read/write/seek operations to the file.

Remarks

None.

Preconditions

A valid file handle has to be passed as input to the function. The file which has to be flushed, has to be present and should have been opened in write mode.

Example

```

SYS_FS_RESULT res;
SYS_FS_HANDLE fileHandle;
const char *buf = "Hello World";
size_t nbytes;
size_t bytes_written;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG", (SYS_FS_FILE_OPEN_WRITE_PLUS));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

```

```

// Write data to the file
bytes_written = SYS_FS_FileWrite((const void *)buf, nbytes, fileHandle);

// Flush the file
res = SYS_FSFileSync(fileHandle);
if( res != SYS_FS_RES_SUCCESS)
{
    // renaming has gone wrong
}

```

Parameters

Parameters	Description
handle	valid file handle

Function

```

SYS_FS_RESULT SYS_FSFileSync
(
    SYS_FS_HANDLE handle
);

```

SYS_FS_FileTell Function

Obtains the file pointer position.

File

`sys_fs.h`

C

```
int32_t SYS_FS_FileTell(SYS_FS_HANDLE handle);
```

Returns

On success returns the current file position. On failure returns -1. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

Obtains the current value of the file position indicator for the file pointed to by handle.

Remarks

None.

Preconditions

A valid file handle must be obtained before performing a file tell.

Example

```

SYS_FS_HANDLE fileHandle;
int32_t tell;

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG" ,
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}
...
...

tell = SYS_FS_FileTell(fileHandle);

```

```

if(tell != -1)
{
    // Success
}

```

Parameters

Parameters	Description
handle	File handle obtained during file Open.

Function

```

int32_t SYS_FS_FileTell
(
    SYS_FS_HANDLE handle
);

```

SYS_FS_FileTestError Function

Checks for errors in the file.

File

[sys_fs.h](#)

C

```
bool SYS_FS_FileTestError(SYS_FS_HANDLE handle);
```

Returns

On success returns false indicating that the file has no errors. On failure returns true. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function checks whether or not file has any errors.

Remarks

None.

Preconditions

A valid file handle must be obtained before passing to the function

Example

```

SYS_FS_HANDLE fileHandle;
bool err;

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG" , (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}
...
...

err = SYS_FS_FileTestError(fileHandle);
if(err == true)
{
    // either file has error, or there
    // was an error in working with the "SYS_FS_FileTestError" function
}

```

Parameters

Parameters	Description
handle	file handle obtained during file Open.

Function

```
bool SYS_FS_FileTestError
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileTruncate Function

Truncates a file

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileTruncate(SYS_FS_HANDLE handle);
```

Returns

SYS_FS_RES_SUCCESS - File truncate operation was successful. SYS_FS_RES_FAILURE - File truncate operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function truncates the file size to the current file read/write pointer. This function has no effect if the file read/write pointer is already pointing to end of the file.

Remarks

None.

Preconditions

A valid file handle has to be passed as input to the function. The file has to be opened in a mode where writes to file is possible (such as read plus or write mode).

Example

```
SYS_FS_HANDLE fileHandle;
size_t nbytes;
size_t bytes_read;
SYS_FS_RESULT res;

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG" ,
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

// Read the file content
nbytes = sizeof(buf);
bytes_read = SYS_FS_FileRead(buf, nbytes, fileHandle);
// Truncate the file
res = SYS_FS_FileTruncate(fileHandle);
if(res != SYS_FS_RES_SUCCESS)
{
    // Truncation failed.
}
```

```
SYS_FS_FileClose(fileHandle);
```

Parameters

Parameters	Description
handle	A valid handle which was obtained while opening the file.

Function

```
SYS_FS_RESULT SYS_FS_FileTruncate
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileWrite Function

Writes data to the file.

File

`sys_fs.h`

C

```
size_t SYS_FS_FileWrite(SYS_FS_HANDLE handle, const void * buf, size_t nbytes);
```

Returns

On success returns the number of bytes written successfully(0 or positive number). On failure returns -1. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function attempts to write nbytes bytes from the buffer pointed to by buf to the file associated with the file handle.

Remarks

None.

Preconditions

A valid file handle must be obtained before writing a file.

Example

```
...
const char *buf = "Hello World";
size_t nbytes;
size_t bytes_written;
SYS_FS_HANDLE fd;
...

bytes_written = SYS_FS_FileWrite(fd, (const void *)buf, nbytes);
...
```

Parameters

Parameters	Description
handle	File handle obtained during file open.
buf	Pointer to buffer from which data is to be written
nbyte	Number of bytes to be written

Function

```
size_t SYS_FS_FileWrite
(

```

```
SYS_FS_HANDLE handle,  
const void *buf,  
size_t nbyte  
);
```

SYS_FS_DirectoryMake Function

Makes a directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirectoryMake(const char* path);
```

Returns

SYS_FS_RES_SUCCESS - Indicates that the creation of the directory was successful. SYS_FS_RES_FAILURE - Indicates that the creation of the directory was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function makes a new directory as per the specified path.

Remarks

None.

Preconditions

The disk has to be mounted before a directory could be made.

Example

```
SYS_FS_RESULT res;  
  
res = SYS_FS_DirectoryMake("Dir1");  
  
if(res == SYS_FS_RES_FAILURE)  
{  
    // Directory make failed  
}
```

Parameters

Parameters	Description
path	Path of the new directory

Function

```
SYS_FS_RESULT SYS_FS_DirectoryMake  
(  
    const char* path  
)
```

SYS_FS_DirOpen Function

Open a directory

File

[sys_fs.h](#)

C

```
SYS_FS_HANDLE SYS_FS_DirOpen(const char* path);
```

Returns

On success a valid handle to the directory will be returned. On failure [SYS_FS_HANDLE_INVALID](#) will be returned. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function opens the requested directory.

Remarks

None

Preconditions

The volume on which the directory is present should be mounted.

Example

```
SYS_FS_HANDLE dirHandle;

dirHandle = SYS_FS_DirOpen("/mnt/myDrive/Dir1");
// For root directory, end with a "/"
// dirHandle = SYS_FS_DirOpen("/mnt/myDrive/");

if(dirHandle != SYS_FS_HANDLE_INVALID)
{
    // Directory open is successful
}
```

Parameters

Parameters	Description
path	Path to the directory along with the volume name. The string of volume and directory name has to be preceded by "/mnt/". Also, the volume name and directory name has to be separated by a slash "/". If the directory specified is only the root directory, the path has to be ended with "/".

Function

```
SYS_FS_HANDLE SYS_FS_DirOpen
(
const char* path
);
```

SYS_FS_DirClose Function

Closes an opened directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirClose(SYS_FS_HANDLE handle);
```

Returns

SYS_FS_RES_SUCCESS - Directory close operation was successful. SYS_FS_RES_FAILURE - Directory close operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function closes a directory that was opened earlier opened with the [SYS_FS_DirOpen](#) function.

Remarks

None.

Preconditions

A valid directory handle must be obtained before closing the directory.

Example

```

SYS_FS_HANDLE dirHandle;

dirHandle = SYS_FS_DirOpen("/mnt/myDrive/Dir1");

if(dirHandle != SYS_FS_HANDLE_INVALID)
{
    // Directory open is successful
}

// Perform required operation on the directory

// Close the directory
if(SYS_FS_DirClose(dirHandle) == SYS_FS_RES_FAILURE)
{
    // Close operation failed.
}

```

Parameters

Parameters	Description
handle	directory handle obtained during directory open.

Function

```

SYS_FS_RESULT SYS_FS_DirClose
(
    SYS_FS_HANDLE handle
);

```

SYS_FS_DirRead Function

Reads the files and directories of the specified directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirRead(SYS_FS_HANDLE handle, SYS_FS_FSTAT * stat);
```

Returns

SYS_FS_RES_SUCCESS - Indicates that the directory read operation was successful. End of the directory condition is indicated by setting the fname and Ifname(if Ifname is used) fields of the [SYS_FS_FSTAT](#) structure to '0'

SYS_FS_RES_FAILURE - Indicates that the directory read operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function reads the files and directories specified in the open directory.

Remarks

None.

Preconditions

A valid directory handle must be obtained before reading a directory.

Example

```

SYS_FS_HANDLE dirHandle;
SYS_FS_FSTAT stat;
char longFileName[300];
uintptr_t longFileSize;

dirHandle = SYS_FS_DirOpen("/mnt/myDrive/Dir1");

if(dirHandle != SYS_FS_HANDLE_INVALID)
{
    // Directory open is successful
}

// If long file name is used, the following elements of the "stat"
// structure needs to be initialized with address of proper buffer.
stat.lfname = longFileName;
stat.lfsize = 300;

if(SYS_FS_DirRead(dirHandle, &stat) == SYS_FS_RES_FAILURE)
{
    // Directory read failed.
}
else
{
    // Directory read succeeded.
    if ((stat.lfname[0] == '0') && (stat.fname[0] == '0'))
    {
        // reached the end of the directory.
    }
    else
    {
        // continue reading the directory.
    }
}
}

```

Parameters

Parameters	Description
handle	Directory handle obtained during directory open.
stat	<p>Pointer to SYS_FS_FSTAT, where the properties of the open directory will be populated after the SYS_FS_DirRead function returns successfully. If LFN is used, then the "Ifname" member of the SYS_FS_FSTAT structure should be initialized with the address of a suitable buffer and the "Ifsize" should be initialized with the size of the buffer. Once the function returns, the buffer whose address is held in "Ifname" will have the file name(long file name)</p> <p>The file system supports 8.3 file name(Short File Name) and also long file name. 8.3 filenames are limited to at most eight characters, followed optionally by a filename extension consisting of a period . and at most three further characters. If the file name fits within the 8.3 limits then generally there will be no valid LFN for it. The stat structure's fname field will contain the SFN and if there is a valid LFN entry for the file then the long file name will be copied into Ifname member of the structure.</p>

Function

```

SYS_FS_RESULT SYS_FS_DirRead
(
    SYS_FS_HANDLE handle,
    SYS_FS_FSTAT *stat
);

```

SYS_FS_DirRewind Function

Rewinds to the beginning of the directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirRewind(SYS_FS_HANDLE handle);
```

Returns

SYS_FS_RES_SUCCESS - Directory rewind operation was successful. SYS_FS_RES_FAILURE - Directory rewind operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function rewinds the directory to the start. Once a search of directory or directory read is completed, the rewind function is used to begin searching the directory from the start.

Remarks

None.

Preconditions

A valid directory handle must be obtained before reading a directory.

Example

```
SYS_FS_HANDLE dirHandle;
SYS_FS_FSTAT stat;
char longFileName[300];
uintptr_t longFileSize;

dirHandle = SYS_FS_DirOpen("/mnt/myDrive/Dir1");

if(dirHandle != SYS_FS_HANDLE_INVALID)
{
    // Directory open is successful
}

// If long file name is used, the following elements of the "stat"
// structure needs to be initialized with address of proper buffer.
stat.lfname = longFileName;
stat.lfsize = 300;

if(SYS_FS_DirRead(dirHandle, &stat) == SYS_FS_RES_FAILURE)
{
    // Directory read operation failed.
}

// Do more search
// Do some more search

// Now, rewind the directory to begin search from start

if(SYS_FS_DirRewind(dirHandle) == SYS_FS_RES_FAILURE)
{
    // Directory rewind failed.
}
```

Parameters

Parameters	Description
handle	directory handle obtained during directory open.

Function

```
SYS_FS_RESULT SYS_FS_DirRewind
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_DirSearch Function

Searches for a file or directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirSearch(SYS_FS_HANDLE handle, const char * name, SYS_FS_FILE_DIR_ATTR
attr, SYS_FS_FSTAT * stat);
```

Returns

SYS_FS_RES_SUCCESS - Indicates that the file or directory was found. The stat parameter will contain information about the file or directory. SYS_FS_RES_FAILURE - Indicates that the file or directory was not found. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function searches for the requested file or directory. The file or directory is specified in the attr parameter, which is of type [SYS_FS_FILE_DIR_ATTR](#).

Remarks

None.

Preconditions

A valid directory handle must be obtained before searching the directory.

Example

```
SYS_FS_HANDLE dirHandle;
SYS_FS_FSTAT stat;
char longFileName[300];
uintptr_t longFileSize;

dirHandle = SYS_FS_DirOpen("/mnt/myDrive/Dir1");

if(dirHandle != SYS_FS_HANDLE_INVALID)
{
    // Directory open is successful
}

// If long file name is used, the following elements of the "stat"
// structure needs to be initialized with address of proper buffer.
stat.lfname = longFileName;
stat.lfsize = 300;

if(SYS_FS_DirSearch(dirHandle, "FIL*.*", SYS_FS_ATTR_ARC, &stat) == SYS_FS_RES_FAILURE)
{
    // Specified file not found
}
else
{
    // File found. Read the complete file name from "stat.lfname" and
    // other file parameters from the "stat" structure
}
```

Parameters

Parameters	Description
handle	directory handle obtained during directory open.
name	name of file or directory needed to be searched. The file name can have wild card entries as follows
?	• - Indicates the rest of the filename or extension can vary (e.g. FILE.*) Indicates that one character in a filename can vary (e.g. F?LE.T?T)
attr	Attribute of the name of type SYS_FS_FILE_DIR_ATTR . This attribute specifies whether to search a file or a directory. Other attribute types could also be specified.
stat	Empty structure of type SYS_FS_FSTAT , where the properties of the file/directory will be populated. If LFN is used, then the "Ifname" member of the SYS_FS_FSTAT structure should be initialized with address of suitable buffer. Also, the "Ifilesize" should be initialized with the size of buffer. Once the function returns, the buffer whose address is held in "Ifname" will have the file name (long file name).

Function

```
SYS_FS_RESULT SYS_FS_DirSearch
(
    SYS_FS_HANDLE handle,
    const char * name,
    SYS_FS_FILE_DIR_ATTR attr,
    SYS_FS_FSTAT *stat
);
```

SYS_FS_DirectoryChange Function

Changes to a the directory specified.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirectoryChange(const char* path);
```

Returns

[SYS_FS_RES_SUCCESS](#) - Indicates that the directory change operation was successful. [SYS_FS_RES_FAILURE](#) - Indicates that the directory change operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function changes the present directory to a new directory.

Remarks

None.

Preconditions

The disk has to be mounted and the directory to be changed must exist.

Example

```
SYS_FS_RESULT res;

res = SYS_FS_DirectoryChange("Dir1");

if(res == SYS_FS_RES_FAILURE)
{
    // Directory change failed
}
```

Parameters

Parameters	Description
path	Path of the directory to be changed to.

Function

```
SYS_FS_RESULT SYS_FS_DirectoryChange
(
const char* path
);
```

SYS_FS_CurrentWorkingDirectoryGet Function

Gets the current working directory

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_CurrentWorkingDirectoryGet(char * buff, uint32_t len);
```

Returns

SYS_FS_RES_SUCCESS - Get current working directory operation was successful. SYS_FS_RES_FAILURE - Get current working directory operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function gets the current working directory path along with the working drive.

Remarks

None.

Preconditions

At least one disk must be mounted.

Example

```
SYS_FS_RESULT res;
char buffer[16];

switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount("/dev/mmcblk0", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
        }
        else
        {
            // Mount was successful. Create a directory.
            AppState = CREATE_DIR;
        }
        break;

    case CREATE_DIR:
        res = SYS_FS_DirectoryMake("Dir1");
        if(res == SYS_FS_RES_FAILURE)
        {
            // Directory creation failed
            AppState = ERROR;
        }
}
```

```

    else
    {
        // Directory creation was successful. Change to the new
        // directory.
        appState = CHANGE_DIR;
    }
    break;
```



```

case CHANGE_DIR:
    res = SYS_FS_DirectoryChange("Dir1");
    if(res == SYS_FS_RES_FAILURE)
    {
        // Directory change failed
        appState = ERROR;
    }
    else
    {
        // Directory change was successful. Get current working
        // directory
        appState = GET_CWD;
    }
    break;
```



```

case GET_CWD:
    res = SYS_FS_CurrentWorkingDirectoryGet(buffer, 15);
    if(res == SYS_FS_RES_FAILURE)
    {
        // Get current directory operation failed
        appState = ERROR;
    }
    break;
```

}

Parameters

Parameters	Description
buff	Pointer to a buffer which will contain the name of the current working directory and drive, once the function completes.
len	Size of the buffer.

Function

```

SYS_FS_RESULT SYS_FS_CurrentWorkingDirectoryGet
(
    char *buff,
    uint32_t len
);
```

SYS_FS_FileDirectoryModeSet Function

Sets the mode for the file or directory.

File

[sys_fs.h](#)

C

```

SYS_FS_RESULT SYS_FS_FileDirectoryModeSet(const char* path, SYS_FS_FILE_DIR_ATTR attr,
SYS_FS_FILE_DIR_ATTR mask);
```

Returns

SYS_FS_RES_SUCCESS - Mode set operation was successful. **SYS_FS_RES_FAILURE** - Mode set operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function sets the mode for a file or directory from the specified list of attributes.

Remarks

None.

Preconditions

The file or directory for which the mode is to be set must exist.

Example

```
// Set read-only flag, clear archive flag and others are retained.
SYS_FS_FileDirectoryModeSet("file.txt", SYS_FS_ATTR_RDO, SYS_FS_ATTR_RDO | SYS_FS_ATTR_ARC);
```

Parameters

Parameters	Description
path	Path for the file/directory, for which the mode is to be set.
attr	Attribute flags to be set in one or more combination of the type SYS_FS_FILE_DIR_ATTR . The specified flags are set and others are cleared.
mask	Attribute mask of type SYS_FS_FILE_DIR_ATTR that specifies which attribute is changed. The specified attributes are set or cleared.

Function

```
SYS_FS_RESULT SYS_FS_FileDirectoryModeSet
(
    const char* path,
    SYS_FS_FILE_DIR_ATTR attr,
    SYS_FS_FILE_DIR_ATTR mask
);
```

SYS_FS_FileDirectoryRemove Function

Removes a file or directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileDirectoryRemove(const char* path);
```

Returns

[SYS_FS_RES_SUCCESS](#) - Indicates that the file or directory remove operation was successful. [SYS_FS_RES_FAILURE](#) - Indicates that the file or directory remove operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function removes a file or directory as specified by the path.

Remarks

None.

Preconditions

- The disk has to be mounted before a directory could be removed.
- The file or directory to be removed has to present.
- The file/sub-directory must not have read-only attribute (AM_RDO), or the function will be rejected with FR_DENIED.
- The sub-directory must be empty and must not be current directory, or the function will be rejected with FR_DENIED.

- The file/sub-directory must not be opened.

Example

```
SYS_FS_RESULT res;

res = SYS_FS_FileDirectoryRemove( "Dir1" );

if(res == SYS_FS_RES_FAILURE)
{
    // Directory remove operation failed
}
//...
//...
```

Parameters

Parameters	Description
path	Path of the File or directory to be removed.

Function

```
SYS_FS_RESULT SYS_FS_FileDirectoryRemove
(
const char* path
);
```

SYS_FS_FileDirectoryRenameMove Function

Renames or moves a file or directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileDirectoryRenameMove( const char * oldPath, const char * newPath );
```

Returns

SYS_FS_RES_SUCCESS - Rename/move operation was successful. SYS_FS_RES_FAILURE - Rename/move operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function renames or moves a file or directory.

Remarks

This function cannot move files/ directory from one drive to another. Do not rename/ move files which are open.

Preconditions

The file or directory to be renamed or moved must exist. This function cannot move files or directories from one drive to another. Do not rename or move files that are open.

Example

```
SYS_FS_RESULT res;

// rename "file.txt" to "renamed_file.txt"
res = SYS_FS_FileDirectoryRenameMove( "file.txt", "renamed_file.txt" );
if( res != SYS_FS_RES_SUCCESS )
{
    // Rename operation failed.
}

// Now, move "renamed_file.txt" inside directory "Dir1"
```

```

res = SYS_FS_FileDirectoryRenameMove( "renamed_file.txt" , "Dir1/renamed_file.txt" );
if( res != SYS_FS_RES_SUCCESS )
{
    // File move operation failed.
}

```

Parameters

Parameters	Description
oldPath	Path for the file/directory, which has to be renamed/moved.
newPath	New Path for the file/directory.

Function

```

SYS_FS_RESULT SYS_FS_FileDirectoryRenameMove
(
    const char *oldPath,
    const char *newPath
);

```

SYS_FS_FileDirectoryTimeSet Function

Sets or changes the time for a file or directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileDirectoryTimeSet(const char* path, SYS_FS_TIME * time);
```

Returns

SYS_FS_RES_SUCCESS - Set time operation was successful. SYS_FS_RES_FAILURE - Set time operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function sets or change the time for a file or directory.

Remarks

None.

Preconditions

The file/directory for which time is to be set must exist.

Example

```

void setTime(void)
{
    SYS_FS_RESULT res;
    SYS_FS_TIME time;

    time.packedTime = 0;

    // All FAT FS times are calculated based on 0 = 1980
    time.discreteTime.year = (2013 - 1980); // Year is 2013
    time.discreteTime.month = 8;           // Month (August)
    time.discreteTime.day = 9;            // Day (9)
    time.discreteTime.hour = 15;          // 3 PM
    time.discreteTime.minute = 06;        // 06 minutes
    time.discreteTime.second = 00;        // 00 seconds

    res = SYS_FS_FileDirectoryTimeSet("file.txt", &time);
    if( res != SYS_FS_RES_SUCCESS)

```

```

    {
        // time change has gone wrong
    }
}

```

Parameters

Parameters	Description
path	A path for the file/directory, for which the time is to be set.
ptr	Pointer to the structure of type SYS_FS_TIME , which contains the time data to be set.

Function

```

SYS\_FS\_RESULT SYS_FS_FileDirectoryTimeSet
(
    const char* path,
    SYS\_FS\_TIME *time
);

```

SYS_FS_Error Function

Returns the last error.

File

[sys_fs.h](#)

C

```
SYS_FS_ERROR SYS_FS_Error();
```

Returns

Error code of type [SYS_FS_ERROR](#).

Description

When a file system operation fails, the application can know the reason of failure by calling the [SYS_FS_Error](#). This function only reports the errors which are not file (or file handle) specific. For example, for functions such as [SYS_FS_Mount](#) and [SYS_FS_FileOpen](#), which do not take handle, any errors happening inside such function calls could be reported using [SYS_FS_Error](#) function. Even for functions, which take handle as its input parameters, the [SYS_FS_Error](#) function can be used to report the type of error for cases where the passed handle itself is invalid.

Remarks

None.

Preconditions

This function has to be called immediately after a failure is observed while doing a file operation. Any subsequent failure will overwrite the cause of previous failure.

Example

```

SYS_FS_HANDLE fileHandle;
SYS_FS_ERROR err;

fileHandle = SYS_FS_FileOpen( "/mnt/myDrive/FILE.JPG",
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle == SYS_FS_HANDLE_INVALID)
{
    // If failure, now know the specific reason for failure
    err = SYS_FS_Error();
}

```

Function

```
SYS_FS_ERROR SYS_FS_Error
(
void
)
```

b) General Operation Functions

SYS_FS_Initialize Function

Initializes the file system abstraction layer (sys_fs layer).

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_Initialize(const void* initData);
```

Returns

SYS_FS_RES_SUCCESS - SYS FS Layer was initialized successfully. SYS_FS_RES_FAILURE - SYS FS Layer initialization failed. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function initializes the abstraction layer (sys_fs layer) and sets up the necessary parameters.

Preconditions

This is the first function to be called during usage of sys_fs. Calling other functions of sys_fs without initializing the sys_fs will cause unpredictable behavior.

Example

```
// This code shows an example of how the SYS FS is initialized
// Only one file system is used

#define SYS_FS_MAX_FILE_SYSTEM_TYPE 1

// Function pointer table for FAT FS
const SYS_FS_FUNCTIONS FatFsFunctions =
{
    .mount      = f_mount,
    .unmount    = f_unmount,
    .open       = f_open,
    .read       = f_read,
    .write      = f_write,
    .close      = f_close,
    .seek       = f_lseek,
    .tell       = f_tell,
    .eof        = f_eof,
    .size       = f_size,
    .fstat      = f_stat,
};

const SYS_FS_REGISTRATION_TABLE sysFSInit [ SYS_FS_MAX_FILE_SYSTEM_TYPE ] =
{
    {
        .nativeFileSystemType = FAT,
        .nativeFileSystemFunctions = &FatFsFunctions
    }
}
```

```

};

SYS_FS_Initialize((const void *)sysFSInit);

```

Parameters

Parameters	Description
initData	Pointer to an array of type SYS_FS_REGISTRATION_TABLE . The number of elements of array is decided by the definition SYS_FS_MAX_FILE_SYSTEM_TYPE . If the application uses one file system (say only FAT FS), SYS_FS_MAX_FILE_SYSTEM_TYPE is defined to be 1. Otherwise, if the application uses 2 file systems (say FAT FS and MPFS2), SYS_FS_MAX_FILE_SYSTEM_TYPE is defined to be 2.

Function

```

SYS_FS_RESULT SYS_FS_Initialize
(
const void* initData
);

```

SYS_FS_Tasks Function

Maintains the File System tasks and functionalities.

File

[sys_fs.h](#)

C

```
void SYS_FS_Tasks();
```

Returns

None.

Description

This function is used to run the various tasks and functionalities of sys_fs layer.

Remarks

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

Preconditions

The [SYS_FS_Initialize](#) routine must have been called before running the tasks.

Example

```

void SYS_Tasks ( void )
{
    SYS_FS_Tasks ();
    // Do other tasks
}

```

Function

```

void SYS_FS_Tasks
(
void
);

```

SYS_FS_Mount Function

Mounts the file system.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_Mount(const char * devName, const char * mountName,
SYS_FS_FILE_SYSTEM_TYPE filesystemtype, unsigned long mountflags, const void * data);
```

Returns

SYS_FS_RES_SUCCESS - Mount was successful. SYS_FS_RES_FAILURE - Mount was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

The mount command attaches the file system specified to a volume. The call to the mount should be non blocking in nature. The application code has to allow the [SYS_FS_Tasks](#) to run periodically while calling the SYS_FS_Mount function. If the SYS_FS_Mount is called in a blocking mode, then the SYS_Tasks() never gets a chance to run and therefore, the media will not be analyzed and finally, the SYS_FS_Mount will never succeed. This will result in a deadlock.

There is no mechanism available for the application to know if the specified volume (devName) is really attached or not. The only available possibility is to keep trying to mount the volume (with the devname), until success is achieved.

It is prudent that the application code implements a time-out mechanism while trying to mount a volume (by calling SYS_FS_Mount). The trial for mount should continue at least 10 times before assuming that the mount will never succeed. This has to be done for every new volume to be mounted.

The standard names for volumes (devName) used in the MPLAB Harmony file system is as follows: NVM - "nvm" "media number" "volume number" SD card - "mmcblk" "media number" "volume number" MSD - "sd" "media number" "volume number"

Where, "media number" a, b, c... depends on the number of the type of connected media, and where, "volume number" 1, 2, 3... depends on the number of partitions in that media.

The convention for assigning names to volumes is further described below with examples:

If a SD card (with four partitions) is attached to the system, and assuming all four partitions are recognized, there will be four devNames:

1. mmcblk1a1
2. mmcblk1a2
3. mmcblk1a3 and
4. mmcblk1a4

Subsequently, if NVM media is attached that has only one partition, the devname will be: nvma1.

Later, if another SD card is attached to the system that has one partition, the devname will be mmcblk1b1.

Finally, there will be six volume names (or devNames), which are available for the application to be mounted and used for the file system.

Remarks

None

Preconditions

The "devName" name for the volume has to be known. The file system type with which each of the volumes are formatted has to be known. Trying to mount a volume with a file system which is different from what the volume is actually formatted, will cause mount failure.

Example

```
switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount("/dev/mmcblk1a1", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
    {
```

```

        // Failure, try mounting again
    }
    else
    {
        // Mount was successful. Do further file operations
        appState = DO_FURTHER_STUFFS;
    }
break;
}

```

Parameters

Parameters	Description
devName	The device name (name of volume) which needs to be mounted. The devName has to be preceded by the string "/dev/".
mountName	Mount name for the device to be mounted. This is a name provided by the user. In future, while accessing the mounted volume (say, during SYS_FS_FileOpen operation), the mountName is used to refer the path for file. The mount name has to be preceded by the string "/mnt/"
filesystemtype	Native file system of SYS_FS_FILE_SYSTEM_TYPE type.
mountflags	Mounting control flags. This parameter is reserved for future enhancements. Therefore, always pass zero.
data	The data argument is interpreted by the different file systems. This parameter is reserved for future enhancements. Therefore, always pass NULL.

Function

```

SYS_FS_RESULT SYS_FS_Mount
(
    const char *devName,
    const char *mountName,
    SYS_FS_FILE_SYSTEM_TYPE filesystemtype,
    unsigned long mountflags,
    const void *data
);

```

SYS_FS_Unmount Function

Unmounts the file system.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_Unmount(const char * mountName);
```

Returns

[SYS_FS_RES_SUCCESS](#) - Unmount was successful. [SYS_FS_RES_FAILURE](#) - Unmount was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function removes (unmounts) the attachment of the volume from the file system.

Preconditions

The volume name has to be known in order to pass as input to Unmount. The specified volume name to be unmounted should have been already mounted.

Example

```
if(SYS_FS_Unmount( "/mnt/myDrive" ) != SYS_FS_RES_SUCCESS)
```

```

{
    // Failure, try unmounting again
}
else
{
    // Unmount was successful.
}

```

Parameters

Parameters	Description
mountName	Mount name for the volume to be unmounted. The mount name has to be preceded by the string "/mnt/".

Function

```

SYS_FS_RESULT SYS_FS_Unmount
(
const char *mountName
);

```

SYS_FS_CurrentDriveGet Function

Gets the current drive

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_CurrentDriveGet(char* buffer);
```

Returns

SYS_FS_RES_SUCCESS - Current drive get operation was successful. SYS_FS_RES_FAILURE - Current drive get operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function gets the present drive being used. The drive information is populated in the buffer.

Remarks

None.

Preconditions

The disk has to be mounted.

Example

```

SYS_FS_RESULT res;
char buffer[255];

res = SYS_FS_CurrentDriveGet(buffer);
if(res == SYS_FS_RES_FAILURE)
{
    // Operation failed.
}

```

Parameters

Parameters	Description
buffer	Pointer to buffer which will hold the name of present drive being used.

Function

[SYS_FS_RESULT SYS_FS_CurrentDriveGet](#)

```

(
char* buffer
);

```

SYS_FS_CurrentDriveSet Function

Sets the drive.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_CurrentDriveSet(const char* path);
```

Returns

SYS_FS_RES_SUCCESS - Current drive set operation was successful. SYS_FS_RES_FAILURE - Current drive set operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function sets the present drive to the one as specified by the path. By default, the drive mounted last becomes the current drive for the system. This is useful for applications where only one drive (volume) is used. In such an application, there is no need to call the SYS_FS_CurrentDriveSet function. However, in the case of an application where there are multiple volumes, the user can select the current drive for the application by calling this function.

Remarks

None.

Preconditions

The disk has to be mounted.

Example

```

SYS_FS_RESULT res;

res = SYS_FS_CurrentDriveSet( "/mnt/myDrive" );
if(res == SYS_FS_RES_FAILURE)
{
    // Drive change failed
}

```

Parameters

Parameters	Description
path	Path for the drive to be set.

Function

```

SYS_FS_RESULT SYS_FS_CurrentDriveSet
(
const char* path
);

```

SYS_FS_DriveLabelGet Function

Gets the drive label.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DriveLabelGet(const char* drive, char * buff, uint32_t * sn);
```

Returns

SYS_FS_RES_SUCCESS - Drive label information retrieval was successful. SYS_FS_RES_FAILURE - Drive label information retrieval was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function gets the label for the drive specified. If no drive is specified, the label for the current drive is obtained.

Remarks

None.

Preconditions

At least one disk must be mounted.

Example

```
SYS_FS_RESULT res;
char buffer[255];
uint32_t serialNo;

switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount( "/dev/mmcblk0", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
        }
        else
        {
            // Mount was successful. Get label now
            AppState = GET_LABEL;
        }
        break;

    case GET_LABEL:
        res = SYS_FS_DriveLabelGet( "/mnt/myDrive", buffer, &serialNo);

        if(res == SYS_FS_RES_FAILURE)
        {
            // Fetching drive label information failed
        }
        //...
        //...
        break;
}
```

Parameters

Parameters	Description
drive	Pointer to buffer which will hold the name of drive being for which the label is requested. If this string is NULL, then the label of the current drive is obtained by using this function.
buff	Buffer which will hold the string of label.
sn	Serial number of the drive. If this information is not needed, it can be set as NULL.

Function

```
SYS_FS_RESULT SYS_FS_DriveLabelGet
(
    const char* drive,
    char *buff,
```

```
uint32_t *sn
);
```

SYS_FS_DriveLabelSet Function

Sets the drive label

File

`sys_fs.h`

C

```
SYS_FS_RESULT SYS_FS_DriveLabelSet(const char * drive, const char * label);
```

Returns

`SYS_FS_RES_SUCCESS` - Drive label set operation was successful. `SYS_FS_RES_FAILURE` - Drive label set operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function sets the label for the drive specified. If no drive is specified, the label for the current drive is set.

Remarks

None.

Preconditions

At least one disk must be mounted.

Example

```
SYS_FS_RESULT res;

switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount("/dev/mmcblk0", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
        }
        else
        {
            // Mount was successful. Get label now
            AppState = GET_LABEL;
        }
        break;

    case GET_LABEL:
        res = SYS_FS_DriveLabelSet("/mnt/myDrive", "MY_LABEL");
        if(res == SYS_FS_RES_FAILURE)
        {
            // Drive label get failed
        }
        //...
        //...
        break;
}
```

Parameters

Parameters	Description
drive	Pointer to string that holds the name of drive being for which the label is to be set. If this string is NULL, the label of the current drive is set by using this function.
label	Pointer to string which contains the label to be set.

Function

```
SYS_FS_RESULT SYS_FS_DriveLabelSet
(
const char* drive,
const char *label
);
```

SYS_FS_DriveFormat Function

Formats a drive.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DriveFormat(const char* drive, SYS_FS_FORMAT fmt, uint32_t clusterSize);
```

Returns

SYS_FS_RES_SUCCESS - Drive format was successful. SYS_FS_RES_FAILURE - Drive format was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function formats a logic drive (create a FAT file system on the logical drive), as per the format specified.

If the logical drive that has to be formatted has been bound to any partition (1-4) by multiple partition feature, the FAT volume is created into the specified partition. In this case, the second argument fmt is ignored. The physical drive must have been partitioned prior to using this function.

Remarks

None.

Preconditions

At least one disk must be mounted. The physical drive must have already been partitioned.

Example

```
SYS_FS_RESULT res;

switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount("/dev/mmcblk0p1", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
        }
        else
        {
            // Mount was successful. Format now.
            AppState = FORMAT_DRIVE;
        }
        break;

    case FORMAT_DRIVE:
        res = SYS_FS_DriveFormat("/mnt/myDrive", SYS_FS_FORMAT_SFD, 0);
        if(res == SYS_FS_RES_FAILURE)
        {
            // Format of the drive failed.
        }
        //...
        break;
}
```

}

Parameters

Parameters	Description
drive	Pointer to buffer which will hold the name of drive being for which the format is to be done. If this string is NULL, then then current drive will be formatted. It is important to end the drive name with a "/".
fmt	Format type.
clusterSize	Cluster size. The value must be sector (size * n), where n is 1 to 128 and power of 2. When a zero is given, the cluster size depends on the volume size.

Function

```
SYS_FS_RESULT SYS_FS_DriveFormat
(
    const char* drive,
    SYS_FS_FORMAT fmt,
    uint32_t clusterSize
);
```

SYS_FS_DrivePartition Function

Partitions a physical drive (media).

File

sys_fs.h

C

```
SYS_FS_RESULT SYS_FS_DrivePartition(const char * path, const uint32_t partition[], void * work);
```

Returns

SYS_FS_RES_SUCCESS - Partition was successful. SYS_FS_RES_FAILURE - Partition was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function partitions a physical drive (media) into requested partition sizes. This function will alter the MBR of the physical drive and make it into multi partitions. Windows operating systems do not support multi partitioned removable media. Maximum 4 partitions can be created on a media.

Remarks

None

Preconditions

Prior to partitioning the media, the media should have a valid MBR and it should be mounted as a volume with the file system.

Example

```
=====
// Initially, consider the case of a SD card that has only one partition.
=====
SYS_FS_RESULT res;

// Following 4 element array specifies the size of 2 partitions as
// 256MB (=524288 sectors). The 3rd and 4th partition are not created
// since, the sizes of those are zero.
uint32_t plist[] = {524288, 524288, 0, 0};

// Work area for function SYS_FS_DrivePartition
char work[FAT_FS_MAX_SS];
```

```

switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount("/dev/mmcblk0", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
        }
        else
        {
            // Mount was successful. Partition now.
            AppState = PARTITION_DRIVE;
        }
        break;

    case PARTITION_DRIVE:
        res = SYS_FS_DrivePartition("/mnt/myDrive", plist, work);
        if(res == SYS_FS_RES_FAILURE)
        {
            // Drive partition went wrong
        }
        else
        {
            // Partition was successful. Power cycle the board so that
            // all partitions are recognized. Then try mounting both
            // partitions.
        }
        break;

    default:
        break;
}

//=====
//The following code is after the SD card is partitioned and then
//powered ON.
//=====

SYS_FS_RESULT res;

switch(appState)
{
    case TRY_MOUNT_1ST_PARTITION:
        if(SYS_FS_Mount("/dev/mmcblk0", "/mnt/myDrive1", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
            AppState = TRY_MOUNT_1ST_PARTITION;
        }
        else
        {
            // Mount was successful. Mount second partition.
            AppState = TRY_MOUNT_2ND_PARTITION;
        }
        break;

    case TRY_MOUNT_2ND_PARTITION:
        if(SYS_FS_Mount("/dev/mmcblk0", "/mnt/myDrive2", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
            AppState = TRY_MOUNT_2ND_PARTITION;
        }
        else
        {
            // Mount was successful. Try formatting first partition.
            AppState = TRY_FORMATTING_1ST_PARTITION;
        }
        break;

    case TRY_FORMATTING_1ST_PARTITION:

```

```

if(SYS_FS_DriveFormat("/mnt/myDrive1/", SYS_FS_FORMAT_FDISK, 0) == SYS_FS_RES_FAILURE)
{
    // Failure
}
else
{
    // Try formating second partitions.
    appState = TRY_FORMATING_2ND_PARTITION;
}

case TRY_FORMATING_2ND_PARTITION:
if(SYS_FS_DriveFormat("/mnt/myDrive2/", SYS_FS_FORMAT_FDISK, 0) == SYS_FS_RES_FAILURE)
{
    // Failure
}
else
{
    // Use both partitions as 2 separate volumes.
}

default:
    break;
}

```

Parameters

Parameters	Description
path	Path to the volume with the volume name. The string of volume name has to be preceded by "/mnt/". Also, the volume name and directory name has to be separated by a slash "/".
partition	Array with 4 items, where each items mentions the sizes of each partition in terms of number of sector. 0th element of array specifies the number of sectors for first partition and 3rd element of array specifies the number of sectors for fourth partition.
work	Pointer to the buffer for function work area. The size must be at least FAT_FS_MAX_SS bytes.

Function

```

SYS_FS_RESULT SYS_FS_DrivePartition
(
const char *path,
const uint32_t partition[],
void * work
);

```

SYS_FS_DriveSectorGet Function

Obtains total number of sectors and number of free sectors for the specified drive.

File

[sys_fs.h](#)

C

```

SYS_FS_RESULT SYS_FS_DriveSectorGet(const char * path, uint32_t * totalSectors, uint32_t * freeSectors);

```

Returns

SYS_FS_RES_SUCCESS - Sector information get operation was successful. SYS_FS_RES_FAILURE - Sector information get operation was unsucessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

Function to obtain the total number of sectors and number of free sectors in a drive (media).

Remarks

None.

Preconditions

The drive for which the information is to be retrieved should be mounted.

Example

```
uint32_t totalSectors, freeSectors;
SYS_FS_RESULT res;

if(SYS_FS_Mount("/dev/mmcblk0", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
{
    // Failure, try mounting again
}
else
{
    // Mount was successful.
    // Do other FS stuffs.
}
// Perform usual FS tasks.
//.....
//.....

// Now, determine the total sectors and free sectors
res = SYS_FS_DriveSectorGet("/mnt/myDrive", &totalSectors, &freeSectors);
if(res == SYS_FS_RES_FAILURE)
{
    //Sector information get operation failed.
}
```

Parameters

Parameters	Description
path	Path to the volume with the volume name. The string of volume name must be preceded by "/mnt/". Also, the volume name and directory name must be separated by a slash "/".
totalSectors	Pointer to a variable passed to the function, which will contain the total number of sectors available in the drive (media).
freeSectors	Pointer to a variable passed to the function, which will contain the free number of sectors available in the drive (media).

Function

```
SYS_FS_RESULT SYS_FS_DriveSectorGet
(
const char* path,
uint32_t *totalSectors,
uint32_t *freeSectors
);
```

SYS_FS_EventHandlerSet Function

Allows a client to identify an event handling function for the file system to call back when mount/unmount operation has completed.

File

[sys_fs.h](#)

C

```
void SYS_FS_EventHandlerSet(const void * eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the File System to call back when mount/unmount operation has completed. The file system will pass mount name back to the client by calling "eventHandler" when AutoMount feature is enabled for File system.

Remarks

On Mount/Un-Mount of a volume all the registered clients will be notified. The client should check if the mount name passed when event handler is called is the one it is expecting and then proceed as demonstrated in above example.

If the client does not want to be notified when the mount/unmount operation has completed, it does not need to register a callback.

This API is Available only when SYS_FS_AUTOMOUNT_ENABLE is set to true.

Preconditions

The [SYS_FS_Initialize\(\)](#) routine must have been called.

Example

```
// Client registers an event handler with file system. This is done once.
SYS_FS_EventHandlerSet(APP_SysFSEventHandler, (uintptr_t)NULL);

// Event Processing Technique. Event is received when operation is done.
void APP_SysFSEventHandler
(
    SYS_FS_EVENT event,
    void* eventData,
    uintptr_t context
)
{
    switch(event)
    {
        case SYS_FS_EVENT_MOUNT:
            if(strcmp((const char *)eventData, "/mnt/myDrive1") == 0)
            {
                gSDCardMountFlag = true;
            }
            else if(strcmp((const char *)eventData, "/mnt/myDrive2") == 0)
            {
                gNVMMountFlag = true;
            }
            break;

        case SYS_FS_EVENT_UNMOUNT:
            if(strcmp((const char *)eventData, "/mnt/myDrive1") == 0)
            {
                gSDCardMountFlag = false;
            }
            else if(strcmp((const char *)eventData, "/mnt/myDrive2") == 0)
            {
                gNVMMountFlag = false;
            }
            break;

        case SYS_FS_EVENT_ERROR:
            break;
    }
}
```

Parameters

Parameters	Description
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 SYS_FS_EventHandlerSet
(
    const void * eventHandler,
    const uintptr_t context
);
```

c) Media Manager Functions

SYS_FS_MEDIA_MANAGER_Tasks Function

Media manager task function.

File

[sys_fs_media_manager.h](#)

C

```
void SYS_FS_MEDIA_MANAGER_Tasks();
```

Returns

None.

Description

This is the media manager task function. This task must be called repeatedly from the main loop.

Preconditions

None

Function

```
void SYS_FS_MEDIA_MANAGER_Tasks
(
    void
);
```

SYS_FS_MEDIA_MANAGER_TransferTask Function

Media manager transfer task function.

File

[sys_fs_media_manager.h](#)

C

```
void SYS_FS_MEDIA_MANAGER_TransferTask(uint8_t mediaIndex);
```

Returns

None.

Description

This is the media manager transfer task function. This task is repeatedly called by the disk io layer of the native file system for driving the current disk read/write operation to completion.

Preconditions

None

Parameters

Parameters	Description
mediaIndex	disk number of the media

Function

```
void SYS_FS_MEDIA_MANAGER_TransferTask
(
    uint8_t mediaIndex
);
```

SYS_FS_MEDIA_MANAGER_Read Function

Gets data from a specific media address.

File

[sys_fs_media_manager.h](#)

C

```
SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE SYS_FS_MEDIA_MANAGER_Read(uint16_t diskNum, uint8_t * destination, uint8_t * source, const uint32_t nBytes);
```

Returns

Buffer handle of type [SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE](#)

Description

This function gets data from a specific address of media. This function is intended to work with NVM media only, which can have byte level addressing. For other media, such as a SD card, byte addressing is not possible and this function will not work. Also, this function is intended to work with the MPFS2 file system only, which uses byte addressing.

Preconditions

None.

Parameters

Parameters	Description
diskNo	media number
destination	pointer to buffer where data to be placed after read
source	pointer from where data to be read
nBytes	Number of bytes to be read

Function

```
SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE SYS_FS_MEDIA_MANAGER_Read
(
    uint16_t diskNo,
    uint8_t* destination,
    uint8_t* source,
    const unsigned int nBytes
);
```

SYS_FS_MEDIA_MANAGER_SectorRead Function

Reads a specified media sector.

File

[sys_fs_media_manager.h](#)

C

```
SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE SYS_FS_MEDIA_MANAGER_SectorRead(uint16_t diskNum, uint8_t* dataBuffer, uint32_t sector, uint32_t numSectors);
```

Returns

Buffer handle of type [SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE](#).

Description

This function reads a specified media (disk) sector. This is the function in the media manager layer. This function in turn calls the specific sector read function from the list of function pointers of the media driver.

Preconditions

None.

Parameters

Parameters	Description
diskNo	Media number
dataBuffer	Pointer to buffer where data to be placed after read
sector	Sector numer to be read
noSectors	Number of sectors to read

Function

```
SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE SYS_FS_MEDIA_MANAGER_SectorRead
(
    uint16_t diskNo,
    uint8_t* dataBuffer,
    uint32_t sector,
    uint32_t noSectors
);
```

SYS_FS_MEDIA_MANAGER_SectorWrite Function

Writes a sector to the specified media.

File[sys_fs_media_manager.h](#)**C**

```
SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE SYS_FS_MEDIA_MANAGER_SectorWrite(uint16_t diskNum, uint32_t
sector, uint8_t * dataBuffer, uint32_t numSectors);
```

Returns

Buffer handle of type [SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE](#).

Description

This function writes to a sector of the specified media (disk). This is the function in the media manager layer. This function in turn calls the specific sector write function from the list of function pointers of the media driver.

Preconditions

None.

Parameters

Parameters	Description
diskNo	media number
sector	Sector # to which data to be written
dataBuffer	pointer to buffer which holds the data to be written
noSectors	Number of sectors to be written

Function

```
SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE SYS_FS_MEDIA_MANAGER_SectorWrite
(
    uint16_t diskNo,
    uint32_t sector,
    uint8_t * dataBuffer,
    uint32_t noSectors
);
```

SYS_FS_MEDIA_MANAGER_Register Function

Function to register media drivers with the media manager.

File[sys_fs_media_manager.h](#)**C**

```
SYS_FS_MEDIA_HANDLE SYS_FS_MEDIA_MANAGER_Register(SYS_MODULE_OBJ obj, SYS_MODULE_INDEX index,
const SYS_FS_MEDIA_FUNCTIONS * mediaFunctions, SYS_FS_MEDIA_TYPE mediaType);
```

Returns

Valid handle of type [SYS_FS_MEDIA_HANDLE](#) on successful registration of the media driver. Invalid handle of type [SYS_FS_MEDIA_HANDLE_INVALID](#) on unsuccessful registration of the media driver.

Description

This function is called by the media driver to register the functionalities with the media manager. For static media, such as NVM or a SD card, the media drivers register with this function at the time of initialization. For dynamic media, such as MSD, this register function is called dynamically, once the MSD media is connected.

Remarks

None

Preconditions

None.

Parameters

Parameters	Description
obj	driver object (of type SYS_MODULE_OBJ , value returned when driver is initialized)
index	driver index (of type SYS_MODULE_INDEX , value passed during driver initialization and opening)
mediaFunctions	List of media driver functions
mediaType	Type of media

Function

```
SYS\_FS\_MEDIA\_HANDLE SYS\_FS\_MEDIA\_MANAGER\_Register
(
    SYS\_MODULE\_OBJ obj,
    SYS\_MODULE\_INDEX index,
    const SYS\_FS\_MEDIA\_FUNCTIONS *mediaFunctions,
    SYS\_FS\_MEDIA\_TYPE mediaType
)
```

[SYS_FS_MEDIA_MANAGER_RegisterTransferHandler](#) Function

Register the event handler for data transfer events.

File

[sys_fs_media_manager.h](#)

C

```
void SYS\_FS\_MEDIA\_MANAGER\_RegisterTransferHandler(const void * eventHandler);
```

Returns

Pointer to the media geometry on Success else NULL.

Description

This function is used to send the command status for the disk operation.

Preconditions

None.

Parameters

Parameters	Description
eventHandler	Event handler pointer.

Function

```
void SYS\_FS\_MEDIA\_MANAGER\_RegisterTransferHandler
(
    const void *eventHandler
);
```

SYS_FS_MEDIA_MANAGER_DeRegister Function**File**[sys_fs_media_manager.h](#)**C**

```
void SYS_FS_MEDIA_MANAGER_DeRegister(SYS_FS_MEDIA_HANDLE handle);
```

Returns

None.

Description

Function called by a media to deregister itself to the media manager. For static media, (like NVM or SD card), this "deregister function" is never called, since static media never gets deregistered once they are initialized. For dynamic media (like MSD), this register function is called dynamically, once the MSD media is connected.

Preconditions

None.

Parameters

Parameters	Description
handle	Handle of type SYS_FS_MEDIA_HANDLE received when the media was registered

Function

```
void SYS_FS_MEDIA_MANAGER_DeRegister
(
    SYS_FS_MEDIA_HANDLE handle
)
```

SYS_FS_MEDIA_MANAGER_AddressGet Function

Gets the starting media address based on a disk number.

File[sys_fs_media_manager.h](#)**C**

```
uintptr_t SYS_FS_MEDIA_MANAGER_AddressGet(uint16_t diskNo);
```

ReturnsMemory address of type `uintptr_t`.**Description**

This function gets the starting address of a media. This function is intended to work only with MPFS2, which does byte addressing and needs a memory address (not disk number).

Preconditions

None.

Parameters

Parameters	Description
diskNo	media number

Function

```
uintptr_t SYS_FS_MEDIA_MANAGER_AddressGet
(
    uint16_t diskNo
);
```

SYS_FS_MEDIA_MANAGER_MediaStatusGet Function

Gets the media status.

File

[sys_fs_media_manager.h](#)

C

```
bool SYS_FS_MEDIA_MANAGER_MediaStatusGet(const char * devName);
```

Returns

Media attach/detach status of type bool.

Description

This function gets the media status. This function is used by higher layers (sys_fs layer) to know the status of the media whether the media is attached or detached.

Preconditions

None.

Parameters

Parameters	Description
*devName	string name of the media

Function

```
bool SYS_FS_MEDIA_MANAGER_MediaStatusGet
(
    const char *devName
);
```

SYS_FS_MEDIA_MANAGER_VolumePropertyGet Function

Gets the volume property.

File

[sys_fs_media_manager.h](#)

C

```
bool SYS_FS_MEDIA_MANAGER_VolumePropertyGet(const char * devName, SYS_FS_VOLUME_PROPERTY * str);
```

Returns

True or false.

Description

This function gets the property of the volume. This function is used by higher layers (sys_fs layer) to know the property of the

volume as specified in the [SYS_FS_VOLUME_PROPERTY](#) structure.

Preconditions

None.

Parameters

Parameters	Description
<code>*devName</code>	String name of the media
<code>*str</code>	Pointer to structure of type SYS_FS_VOLUME_PROPERTY

Function

```
bool SYS_FS_MEDIA_MANAGER_VolumePropertyGet
(
    const char *devName
    SYS\_FS\_VOLUME\_PROPERTY *str
);
```

SYS_FS_MEDIA_MANAGER_CommandStatusGet Function

Gets the command status.

File

[sys_fs_media_manager.h](#)

C

```
SYS_FS_MEDIA_COMMAND_STATUS SYS\_FS\_MEDIA\_MANAGER\_CommandStatusGet(uint16_t diskNo,
    SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE commandHandle);
```

Returns

Command status of type [SYS_FS_MEDIA_COMMAND_STATUS](#).

Description

This function gets the command status. The sector read and sector write are non-blocking functions. Therefore, this interface is provided where the code should periodically poll for the buffer status. If status is completed, the read/write operation is considered to be complete.

Preconditions

None.

Parameters

Parameters	Description
<code>diskNo</code>	media number
<code>bufferHandle</code>	the command handle which was obtained during sector read/ write

Function

```
SYS\_FS\_MEDIA\_COMMAND\_STATUS SYS\_FS\_MEDIA\_MANAGER\_CommandStatusGet
(
    uint16_t diskNo,
    SYS\_FS\_MEDIA\_BLOCK\_COMMAND\_HANDLE bufferHandle
)
```

SYS_FS_MEDIA_MANAGER_GetMediaGeometry Function

Gets the media geometry information.

File

[sys_fs_media_manager.h](#)

C

```
SYS_FS_MEDIA_GEOMETRY * SYS_FS_MEDIA_MANAGER_GetMediaGeometry(uint16_t diskNum);
```

Returns

Pointer to the media geometry on Success else NULL.

Description

This function gets the media geometry information.

Preconditions

None.

Parameters

Parameters	Description
diskNum	Media disk number.

Function

```
SYS_FS_MEDIA_GEOMETRY * SYS_FS_MEDIA_MANAGER_GetMediaGeometry
(
    uint16_t diskNum
);
```

SYS_FS_MEDIA_MANAGER_EventHandlerSet Function

Register the event handler for Mount/Un-Mount events.

File

[sys_fs_media_manager.h](#)

C

```
void SYS_FS_MEDIA_MANAGER_EventHandlerSet(const void * eventHandler, const uintptr_t context);
```

Description

This function is used to register a FS client event handler for notifying the Mount/Un-Mount events when AutoMount feature is enabled for File system.

On Mount/Un-Mount of a volume all the registered clients will be notified. The client should check if the mount name passed when event handler is called is the one it is expecting and then proceed.

Remarks

This API should not be called directly from Application. Application should use [SYS_FS_EventHandlerSet\(\)](#) instead.

This API is Available only when [SYS_FS_AUTOMOUNT_ENABLE](#) is set to true.

See [sys_fs.h](#) for usage information.

Parameters

Parameters	Description
eventHandler	FS Client event handler pointer
context	FS Client context

Function

```
void SYS_FS_MEDIA_MANAGER_EventHandlerSet
(
    const void * eventHandler,
    const uintptr_t context
);
```

d) File System Data Types and Constants

SYS_FS_ERROR Enumeration

Lists the various error cases.

File

[sys_fs.h](#)

C

```
typedef enum {
    SYS_FS_ERROR_OK = 0,
    SYS_FS_ERROR_DISK_ERR,
    SYS_FS_ERROR_INT_ERR,
    SYS_FS_ERROR_NOT_READY,
    SYS_FS_ERROR_NO_FILE,
    SYS_FS_ERROR_NO_PATH,
    SYS_FS_ERROR_INVALID_NAME,
    SYS_FS_ERROR_DENIED,
    SYS_FS_ERROR_EXIST,
    SYS_FS_ERROR_INVALID_OBJECT,
    SYS_FS_ERROR_WRITE_PROTECTED,
    SYS_FS_ERROR_INVALID_DRIVE,
    SYS_FS_ERROR_NOT_ENABLED,
    SYS_FS_ERROR_NO_FILESYSTEM,
    SYS_FS_ERROR_FORMAT_ABORTED,
    SYS_FS_ERROR_TIMEOUT,
    SYS_FS_ERROR_LOCKED,
    SYS_FS_ERROR_NOT_ENOUGH_CORE,
    SYS_FS_ERROR_TOO_MANY_OPEN_FILES,
    SYS_FS_ERROR_INVALID_PARAMETER,
    SYS_FS_ERROR_NOT_ENOUGH_FREE_VOLUME,
    SYS_FS_ERROR_FS_NOT_SUPPORTED,
    SYS_FS_ERROR_FS_NOT_MATCH_WITH_VOLUME,
    SYS_FS_ERROR_NOT_SUPPORTED_IN_NATIVE_FS
} SYS_FS_ERROR;
```

Members

Members	Description
SYS_FS_ERROR_OK = 0	Success
SYS_FS_ERROR_DISK_ERR	(1) A hard error occurred in the low level disk I/O layer
SYS_FS_ERROR_INT_ERR	(2) Assertion failed

SYS_FS_ERROR_NOT_READY	(3) The physical drive cannot work
SYS_FS_ERROR_NO_FILE	(4) Could not find the file
SYS_FS_ERROR_NO_PATH	(5) Could not find the path
SYS_FS_ERROR_INVALID_NAME	(6) The path name format is invalid
SYS_FS_ERROR_DENIED	(7) Access denied due to prohibited access or directory full
SYS_FS_ERROR_EXIST	(8) Access denied due to prohibited access
SYS_FS_ERROR_INVALID_OBJECT	(9) The file/directory object is invalid
SYS_FS_ERROR_WRITE_PROTECTED	(10) The physical drive is write protected
SYS_FS_ERROR_INVALID_DRIVE	(11) The logical drive number is invalid
SYS_FS_ERROR_NOT_ENABLED	(12) The volume has no work area
SYS_FS_ERROR_NO_FILESYSTEM	(13) There is no valid volume
SYS_FS_ERROR_FORMAT_ABORTED	(14) The Format() aborted due to any parameter error
SYS_FS_ERROR_TIMEOUT	(15) Could not get a grant to access the volume within defined period
SYS_FS_ERROR_LOCKED	(16) The operation is rejected according to the file sharing policy
SYS_FS_ERROR_NOT_ENOUGH_CORE	(17) LFN working buffer could not be allocated
SYS_FS_ERROR_TOO_MANY_OPEN_FILES	(18) Number of open files
SYS_FS_ERROR_INVALID_PARAMETER	(19) Given parameter is invalid
SYS_FS_ERROR_NOT_ENOUGH_FREE_VOLUME	(20) Too many mounts requested. Not enough free volume available
SYS_FS_ERROR_FS_NOT_SUPPORTED	(21) Requested native file system is not supported
SYS_FS_ERROR_FS_NOT_MATCH_WITH_VOLUME	(22) Requested native file system does not match the format of volume
SYS_FS_ERROR_NOT_SUPPORTED_IN_NATIVE_FS	(23) Function not supported in native file system layer

Description

File Error enumeration

This enumeration lists the various error cases. When the application calls for a file system function which has a return type of [SYS_FS_RESULT](#) and if the return value is [SYS_FS_RES_FAILURE](#), the application can know the specific reason for failure by calling the [SYS_FS_FileError](#) function. The return value of [SYS_FS_FileError](#) function will be one of the enumeration of type [SYS_FS_ERROR](#).

Remarks

None.

SYS_FS_FILE_SEEK_CONTROL Enumeration

Lists the various modes of file seek.

File

[sys_fs.h](#)

C

```
typedef enum {
    SYS_FS_SEEK_SET,
    SYS_FS_SEEK_CUR,
    SYS_FS_SEEK_END
} SYS_FS_FILE_SEEK_CONTROL;
```

Members

Members	Description
SYS_FS_SEEK_SET	Set file offset to input number of bytes from the start of file
SYS_FS_SEEK_CUR	Set file offset to its current location plus input number of bytes
SYS_FS_SEEK_END	Set file offset to size of the file plus input number of bytes

Description

File Seek control

This enumeration lists the various modes of file seek. When the application calls the [SYS_FS_FileSeek](#) function, it specifies the kind of seek that needs to be performed.

Remarks

None.

SYS_FS_FSTAT Structure

File System status

File

[sys_fs.h](#)

C

```
typedef struct {
    uint32_t fsize;
    uint16_t fdate;
    uint16_t ftime;
    uint8_t fattrib;
    char fname[13];
    char * lfname;
    uint32_t lfsiz;
} SYS_FS_FSTAT;
```

Members

Members	Description
uint32_t fsize;	File size
uint16_t fdate;	Last modified date
uint16_t ftime;	Last modified time
uint8_t fattrib;	Attribute
char fname[13];	Short file name (8.3 format)
char * lfname;	Pointer to the LFN buffer
uint32_t lfsiz;	Size of LFN buffer in TCHAR

Description

SYS FS File status structure

This structure holds the various status of a file. The [SYS_FS_FileStat](#) () populates the contents of this structure.

Remarks

None.

SYS_FS_FUNCTIONS Structure

SYS FS Function signature structure for native file systems.

File

[sys_fs.h](#)

C

```
typedef struct {
    int (* mount)(uint8_t vol);
    int (* unmount)(uint8_t vol);
    int (* open)(uintptr_t handle, const char* path, uint8_t mode);
}
```

```

int (* read)(uintptr_t fp, void* buff, uint32_t btr, uint32_t *br);
int (* write)(uintptr_t fp, const void* buff, uint32_t btw, uint32_t* bw);
int (* close)(uintptr_t fp);
int (* seek)(uintptr_t handle, uint32_t offset);
uint32_t (* tell)(uintptr_t handle);
bool (* eof)(uintptr_t handle);
uint32_t (* size)(uintptr_t handle);
int (* fstat)(const char* path, uintptr_t fno);
int (* mkdir)(const char *path);
int (* chdir)(const char *path);
int (* remove)(const char *path);
int (* getlabel)(const char *path, char *buff, uint32_t *sn);
int (* setlabel)(const char *label);
int (* truncate)(uintptr_t handle);
int (* currWD)(char* buff, uint32_t len);
int (* chdrive)(uint8_t drive);
int (* chmode)(const char* path, uint8_t attr, uint8_t mask);
int (* chtime)(const char* path, uintptr_t ptr);
int (* rename)(const char *oldPath, const char *newPath);
int (* sync)(uintptr_t fp);
char * (* getstrn)(char* buff, int len, uintptr_t handle);
int (* putchr)(char c, uintptr_t handle);
int (* putstrn)(const char* str, uintptr_t handle);
int (* formattedprint)(uintptr_t handle, const char *str, ...);
bool (* testerror)(uintptr_t handle);
int (* formatDisk)(uint8_t vol, uint8_t sfd, uint32_t au);
int (* openDir)(uintptr_t handle, const char *path);
int (* readDir)(uintptr_t handle, uintptr_t stat);
int (* closeDir)(uintptr_t handle);
int (* partitionDisk)(uint8_t pdrv, const uint32_t szt[], void* work);
int (* getCluster)(const char *path, uint32_t *tot_sec, uint32_t *free_sec);
} SYS_FS_FUNCTIONS;

```

Members

Members	Description
int (* mount)(uint8_t vol);	Function pointer of native file system for mounting a volume
int (* umount)(uint8_t vol);	Function pointer of native file system for unmounting a volume
int (* open)(uintptr_t handle, const char * path, uint8_t mode);	Function pointer of native file system for opening a file
int (* read)(uintptr_t fp, void * buff, uint32_t btr, uint32_t *br);	Function pointer of native file system for reading a file
int (* write)(uintptr_t fp, const void * buff, uint32_t btw, uint32_t* bw);	Function pointer of native file system for writing to a file
int (* close)(uintptr_t fp);	Function pointer of native file system for closing a file
int (* seek)(uintptr_t handle, uint32_t offset);	Function pointer of native file system for moving the file pointer by <ul style="list-style-type: none"> desired offset
uint32_t (* tell)(uintptr_t handle);	Function pointer of native file system for finding the position of the <ul style="list-style-type: none"> file pointer
bool (* eof)(uintptr_t handle);	Function pointer of native file system to check if the end of file is <ul style="list-style-type: none"> reached
uint32_t (* size)(uintptr_t handle);	Function pointer of native file system to know the size of file
int (* fstat)(const char * path, uintptr_t fno);	Function pointer of native file system to know the status of file
int (* mkdir)(const char *path);	Function pointer of native file system to create a directory
int (* chdir)(const char *path);	Function pointer of native file system to change a directory
int (* remove)(const char *path);	Function pointer of native file system to remove a file or directory
int (* getlabel)(const char *path, char *buff, uint32_t *sn);	Function pointer of native file system to get the volume label
int (* setlabel)(const char *label);	Function pointer of native file system to set the volume label
int (* truncate)(uintptr_t handle);	Function pointer of native file system to truncate the file

int (* currWD)(char* buff, uint32_t len);	Function pointer of native file system to obtain the current working • directory
int (* chdrive)(uint8_t drive);	Function pointer of native file system to set the current drive
int (* chmode)(const char* path, uint8_t attr, uint8_t mask);	Function pointer of native file system to change the attribute for file • or directory
int (* chtime)(const char* path, uintptr_t ptr);	Function pointer of native file system to change the time for a file or • directory
int (* rename)(const char *oldPath, const char *newPath);	Function pointer of native file system to rename a file or directory
int (* sync)(uintptr_t fp);	Function pointer of native file system to flush file
char * (* getstrn)(char* buff, int len, uintptr_t handle);	Function pointer of native file system to read a string from a file
int (* putchr)(char c, uintptr_t handle);	Function pointer of native file system to write a character into a file
int (* putstrn)(const char* str, uintptr_t handle);	Function pointer of native file system to write a string into a file
int (* formattedprint)(uintptr_t handle, const char *str, ...);	Function pointer of native file system to print a formatted string to • file
bool (* testerror)(uintptr_t handle);	Function pointer of native file system to test an error in a file
int (* formatDisk)(uint8_t vol, uint8_t sfd, uint32_t au);	Function pointer of native file system to format a disk
int (* openDir)(uintptr_t handle, const char *path);	Function pointer of native file system to open a directory
int (* readDir)(uintptr_t handle, uintptr_t stat);	Function pointer of native file system to read a directory
int (* closeDir)(uintptr_t handle);	Function pointer of native file system to close an opened directory
int (* partitionDisk)(uint8_t pdrv, const uint32_t szt[], void* work);	Function pointer of native file system to partition a physical drive
int (* getCluster)(const char *path, uint32_t *tot_sec, uint32_t *free_sec);	Function pointer of native file system to get total sectors and free • sectors

Description

SYS FS Function signature structure for native file systems

The SYS FS layer supports functions from each native file system layer. This structure specifies the signature for each function from native file system (parameter that needs to be passed to each function and return type for each function). If a new native file system is to be integrated with the SYS FS layer, the functions should follow the signature.

The structure of function pointer for the two native file systems: FAT FS and MPFS2 is already provided in the respective source files for the native file system. Hence the following structure is not immediately useful for the user. But the explanation for the structure is still provided for advanced users who would wish to integrate a new native file system to the MPLAB Harmony File System framework.

Remarks

None.

SYS_FS_REGISTRATION_TABLE Structure

The sys_fs layer has to be initialized by passing this structure with suitably initialized members.

File

[sys_fs.h](#)

C

```
typedef struct {
    SYS_FS_FILE_SYSTEM_TYPE nativeFileSystemType;
    const SYS_FS_FUNCTIONS * nativeFileSystemFunctions;
} SYS_FS_REGISTRATION_TABLE;
```

Members

Members	Description
SYS_FS_FILE_SYSTEM_TYPE nativeFileSystemType;	Native file system of type SYS_FS_FILE_SYSTEM_TYPE
const SYS_FS_FUNCTIONS * nativeFileSystemFunctions;	Pointer to the structure of type SYS_FS_FUNCTIONS which has the list of

Description

SYS_FS_REGISTRATION_TABLE structure

When the SYS FS layer is initialized, it has to know the type of native file system it has to support and the list of functions for native file system. The members of this structure can be initialized with suitable values and then passed on to [SYS_FS_Initialize](#) initialization function. Please refer to the example code provided for [SYS_FS_Initialize](#).

Remarks

None.

SYS_FS_RESULT Enumeration

Lists the various results of a file operation.

File

[sys_fs.h](#)

C

```
typedef enum {
    SYS_FS_RES_SUCCESS = 0,
    SYS_FS_RES_FAILURE = -1
} SYS_FS_RESULT;
```

Members

Members	Description
SYS_FS_RES_SUCCESS = 0	Operation succeeded
SYS_FS_RES_FAILURE = -1	Operation failed

Description

File operation result enum

This enumeration lists the various results of a file operation. When a file operation function is called from the application, and if the return type of the function is SYS_FS_RESULT, then the enumeration below specifies the possible values returned by the function.

Remarks

None.

SYS_FS_FILE_OPEN_ATTRIBUTES Enumeration

Lists the various attributes (modes) in which a file can be opened.

File

[sys_fs.h](#)

C

```
typedef enum {
    SYS_FS_FILE_OPEN_READ = 0,
    SYS_FS_FILE_OPEN_WRITE,
    SYS_FS_FILE_OPEN_APPEND,
    SYS_FS_FILE_OPEN_READ_PLUS,
```

```

SYS_FS_FILE_OPEN_WRITE_PLUS,
SYS_FS_FILE_OPEN_APPEND_PLUS
} SYS_FS_FILE_OPEN_ATTRIBUTES;

```

Members

Members	Description
SYS_FS_FILE_OPEN_READ = 0	reading the file = possible, if file exists. reading the file = file open returns error, if file does not exist. writing to the file = not possible. Write operation returns error
SYS_FS_FILE_OPEN_WRITE	reading the file = not possible. Read operation returns error. writing to the file = possible. If file exists, write happens from the beginning of the file, overwriting the existing content of the file. writing to the file = If file does not exist, a new file will be created and data will be written into the newly created file.
SYS_FS_FILE_OPEN_APPEND	reading the file = not possible. Read operation returns error writing to the file = possible. If file exists, write happens from the end of the file, preserving the existing content of the file. writing to the file = If file does not exist, a new file will be created and data will be written into the newly created file.
SYS_FS_FILE_OPEN_READ_PLUS	reading the file = possible, if file exists. reading the file = file open returns error, if file does not exist. writing to the file = possible, if file exists, starting from the beginning of the file (overwriting). writing to the file = file open returns error, if file does not exist.
SYS_FS_FILE_OPEN_WRITE_PLUS	reading the file = possible, if file exists. reading the file = If file does not exist, a new file will be created. writing to the file = possible. If file exists, write happens from the beginning of the file, overwriting the existing content of the file. writing to the file = If file does not exist, a new file will be created and data will be written into the newly created file.
SYS_FS_FILE_OPEN_APPEND_PLUS	reading the file = possible, if file exists. File read pointer will be moved to end of the file in this mode. reading the file = If file does not exist, a new file will be created. writing to the file = possible. If file exists, write happens from the end of the file, preserving the existing content of the file. writing to the file = If file does not exist, a new file will be created and data will be written into the newly created file.

Description

File open attributes

This enumeration lists the various attributes (modes) in which a file can be opened.

Remarks

None.

FAT_FS_MAX_LFN Macro

Maximum length of the Long File Name.

File

[sys_fs.h](#)

C

```
#define FAT_FS_MAX_LFN 255
```

Description

FAT File System LFN (Long File Name) max length

Defines the maximum length of file name during LFN selection. Set the value to 255.

Remarks

None.

FAT_FS_MAX_SS Macro

Lists the definitions for FAT file system sector size.

File

[sys_fs.h](#)

C

```
#define FAT_FS_MAX_SS 512
```

Description

FAT File System Sector size

Maximum sector size to be handled. Always set the value of sector size to 512

Remarks

None.

FAT_FS_USE_LFN Macro

Lists the definitions for FAT file system LFN selection.

File

[sys_fs.h](#)

C

```
#define FAT_FS_USE_LFN 1
```

Description

FAT File System LFN (long file name) selection

The FAT_FS_USE_LFN option switches the LFN support. Set the value to 1.

Remarks

None.

SYS_FS_FILE_SYSTEM_TYPE Enumeration

Enumerated data type identifying native file systems supported.

File

[sys_fs.h](#)

C

```
typedef enum {
    UNSUPPORTED_FS = 0,
    FAT,
    MPFS2
} SYS_FS_FILE_SYSTEM_TYPE;
```

Members

Members	Description
UNSUPPORTED_FS = 0	Unsupported File System
FAT	FAT FS native File system
MPFS2	MPFS2 native File system

Description

File System type

These enumerated values identify the native file system supported by the SYS FS.

Remarks

None.

SYS_FS_HANDLE Type

This type defines the file handle.

File

[sys_fs.h](#)

C

```
typedef uintptr_t SYS_FS_HANDLE;
```

Description

SYS FS File Handle

This type defines the file handle. File handle is returned by the File Open function on successful operation.

Remarks

None.

SYS_FS_HANDLE_INVALID Macro

Invalid file handle

File

[sys_fs.h](#)

C

```
#define SYS_FS_HANDLE_INVALID ((SYS_FS_HANDLE)(-1))
```

Description

SYS FS File Invalid Handle

This value defines the invalid file handle. Invalid file handle is returned on an unsuccessful File Open operation.

Remarks

None.

SYS_FS_FILE_DIR_ATTR Enumeration

Enumerated data type identifying the various attributes for file/directory.

File

[sys_fs.h](#)

C

```
typedef enum {
    SYS_FS_ATTR_RDO = 0x01,
    SYS_FS_ATTR_HID = 0x02,
    SYS_FS_ATTR_SYS = 0x04,
    SYS_FS_ATTR_VOL = 0x08,
```

```

SYS_FS_ATTR_LFN = 0x0F,
SYS_FS_ATTR_DIR = 0x10,
SYS_FS_ATTR_ARC = 0x20,
SYS_FS_ATTR_MASK = 0x3F
} SYS_FS_FILE_DIR_ATTR;

```

Members

Members	Description
SYS_FS_ATTR_RDO = 0x01	Read only
SYS_FS_ATTR_HID = 0x02	Hidden
SYS_FS_ATTR_SYS = 0x04	System
SYS_FS_ATTR_VOL = 0x08	Volume label
SYS_FS_ATTR_LFN = 0x0F	LFN entry
SYS_FS_ATTR_DIR = 0x10	Directory
SYS_FS_ATTR_ARC = 0x20	Archive
SYS_FS_ATTR_MASK = 0x3F	Mask of defined bits

Description

File or directory attribute

These enumerated values are the possible attributes for a file or directory.

Remarks

None.

SYS_FS_TIME Union

The structure to specify the time for a file or directory.

File

[sys_fs.h](#)

C

```

typedef union {
    struct discreteTime {
        unsigned second : 5;
        unsigned minute : 6;
        unsigned hour : 5;
        unsigned day : 5;
        unsigned month : 4;
        unsigned year : 7;
    }
    struct timeDate {
        uint16_t time;
        uint16_t date;
    }
    uint32_t packedTime;
} SYS_FS_TIME;

```

Members

Members	Description
unsigned second : 5;	Second / 2 (0..29)
unsigned minute : 6;	Minute (0..59)
unsigned hour : 5;	Hour (0..23)
unsigned day : 5;	Day in month(1..31)
unsigned month : 4;	Month (1..12)
unsigned year : 7;	Year from 1980 (0..127)

uint16_t time;	Time (hour, min, seconds)
uint16_t date;	Date (year, month, day)
uint32_t packedTime;	Combined time information in a 32-bit value

Description

SYS FS File time structure

This structure holds the date and time to be used to set for a file or directory.

bits 31-25: Year from 1980 (0..127) bits 24-21: Month (1..12) bits 20-16: Day in month(1..31) bits 15-11: Hour (0..23) bits 10-5 : Minute (0..59) bits 4-0 : Seconds / 2 (0..29)

Remarks

None.

SYS_FS_FORMAT Enumeration

Specifies the partitioning rule.

File

[sys_fs.h](#)

C

```
typedef enum {
    SYS_FS_FORMAT_FDISK = 0,
    SYS_FS_FORMAT_SFD = 1
} SYS_FS_FORMAT;
```

Members

Members	Description
SYS_FS_FORMAT_FDISK = 0	Format disk with multiple partition
SYS_FS_FORMAT_SFD = 1	Format disk with single partition

Description

File formating partition rule

This type specifies the partitioning rule. When SYS_FS_FORMAT_FDISK format is specified, a primary partition occupying the entire disk space is created and then an FAT volume is created on the partition. When SYS_FS_FORMAT_SFD format is specified, the FAT volume starts from the first sector of the physical drive.

The SYS_FS_FORMAT_FDISK partitioning is usually used for hard disk, MMC, SDC, CFC and U Disk. It can divide a physical drive into one or more partitions with a partition table on the MBR. However Windows does not support multiple partition on the removable media. The SYS_FS_FORMAT_SFD is non-partitioned method. The FAT volume starts from the first sector on the physical drive without partition table. It is usually used for floppy disk, micro drive, optical disk, and super-floppy media.

SYS_FS_EVENT Enumeration

Identifies the possible file system events.

File

[sys_fs.h](#)

C

```
typedef enum {
    SYS_FS_EVENT_MOUNT,
    SYS_FS_EVENT_UNMOUNT,
    SYS_FS_EVENT_ERROR
} SYS_FS_EVENT;
```

Members

Members	Description
SYS_FS_EVENT_MOUNT	Media has been mounted successfully.
SYS_FS_EVENT_UNMOUNT	Media has been unmounted successfully.
SYS_FS_EVENT_ERROR	There was an error during the operation

Description

SYS FS Media Events

This enumeration identifies the possible events that can result from a file system.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the file system by setting the event handler when media mount or unmount is completed.

SYS_FS_EVENT_HANDLER Type

Pointer to the File system Handler function.

File

[sys_fs.h](#)

C

```
typedef void (* SYS_FS_EVENT_HANDLER)(SYS_FS_EVENT event, void* eventData, uintptr_t context);
```

Returns

None.

Description

File System Event Handler function pointer

This data type defines the required function signature for the file system 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 call backs from the file system.

Remarks

None.

Parameters

Parameters	Description
event	Identifies the type of event
eventData	Handle returned from the media operation requests
context	Value identifying the context of the application that registered the event handling function

SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID Macro

Defines the invalid media block command handle.

File

[sys_fs_media_manager.h](#)

C

```
#define SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID SYS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID
```

Description

Invalid media block command handle

This value defines invalid handle for the media block command. It is returned by read/write/erase routines when the request could not be taken.

Remarks

None.

SYS_FS_MEDIA_HANDLE_INVALID Macro

Defines the invalid media handle.

File

[sys_fs_media_manager.h](#)

C

```
#define SYS_FS_MEDIA_HANDLE_INVALID DRV_HANDLE_INVALID
```

Description

Invalid media handle

This value defines invalid media handle. It is returned when the media registration is not successful.

Remarks

None.

SYS_FS_MEDIA_BLOCK_EVENT Enumeration

Identifies the possible events that can result from a request.

File

[sys_fs_media_manager.h](#)

C

```
typedef enum {
    SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE = SYS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE,
    SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR = SYS_MEDIA_EVENT_BLOCK_COMMAND_ERROR
} SYS_FS_MEDIA_BLOCK_EVENT;
```

Members

Members	Description
SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE = SYS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE	Block operation has been completed successfully.
SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR = SYS_MEDIA_EVENT_BLOCK_COMMAND_ERROR	There was an error during the block operation

Description

SYS FS Media Events

This enumeration identifies the possible events that can result from a media.

Remarks

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

SYS_FS_MEDIA_COMMAND_STATUS Enumeration

The enumeration for status of buffer

File

[sys_fs_media_manager.h](#)

C

```
typedef enum {
    SYS_FS_MEDIA_COMMAND_COMPLETED = SYS_MEDIA_COMMAND_COMPLETED,
    SYS_FS_MEDIA_COMMAND_QUEUED = SYS_MEDIA_COMMAND_QUEUED,
    SYS_FS_MEDIA_COMMAND_IN_PROGRESS = SYS_MEDIA_COMMAND_IN_PROGRESS,
    SYS_FS_MEDIA_COMMAND_UNKNOWN = SYS_MEDIA_COMMAND_UNKNOWN
} SYS_FS_MEDIA_COMMAND_STATUS;
```

Members

Members	Description
SYS_FS_MEDIA_COMMAND_COMPLETED = SYS_MEDIA_COMMAND_COMPLETED	Done OK and ready
SYS_FS_MEDIA_COMMAND_QUEUED = SYS_MEDIA_COMMAND_QUEUED	Scheduled but not started
SYS_FS_MEDIA_COMMAND_IN_PROGRESS = SYS_MEDIA_COMMAND_IN_PROGRESS	Currently being in transfer
SYS_FS_MEDIA_COMMAND_UNKNOWN = SYS_MEDIA_COMMAND_UNKNOWN	Unknown buffer

Description

Status of buffer

This enumeration contains the various status of buffer.

Remarks

None.

SYS_FS_MEDIA_FUNCTIONS Structure

Structure of function pointers for media driver

File

[sys_fs_media_manager.h](#)

C

```
typedef struct {
    bool (* mediaStatusGet)(DRV_HANDLE handle);
    SYS_FS_MEDIA_GEOMETRY * (* mediaGeometryGet)(const DRV_HANDLE handle);
    void (* sectorRead)(DRV_HANDLE clientHandle, SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE * commandHandle, void * buffer, uint32_t blockStart, uint32_t nBlock);
    void (* sectorWrite)(const DRV_HANDLE handle, SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE * commandHandle, void * sourceBuffer, uint32_t blockStart, uint32_t nBlock);
    void (* eventHandlerset)(DRV_HANDLE handle, const void * eventHandler, const uintptr_t context);
    SYS_FS_MEDIA_COMMAND_STATUS (* commandStatusGet)(DRV_HANDLE handle, SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE commandHandle);
    void (* Read)(DRV_HANDLE clientHandle, SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE * commandHandle, void * buffer, uint32_t blockStart, uint32_t nBlock);
    uintptr_t (* addressGet)(const DRV_HANDLE hClient);
    void (* erase)(const DRV_HANDLE handle, SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE * commandHandle, uint32_t blockStart, uint32_t nBlock);
}
```

```

DRV_HANDLE (* open)(SYS_MODULE_INDEX index, DRV_IO_INTENT intent);
void (* close)(DRV_HANDLE client);
void (* tasks)(SYS_MODULE_OBJ obj);
} SYS_FS_MEDIA_FUNCTIONS;

```

Members

Members	Description
bool (* mediaStatusGet)(DRV_HANDLE handle);	To obtains status of media
SYS_FS_MEDIA_GEOMETRY * (* mediaGeometryGet)(const DRV_HANDLE handle);	Function to get media geometry
void (* sectorRead)(DRV_HANDLE clientHandle, SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE * commandHandle, void * buffer, uint32_t blockStart, uint32_t nBlock);	Function for sector read
void (* sectorWrite)(const DRV_HANDLE handle, SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE * commandHandle, void * sourceBuffer, uint32_t blockStart, uint32_t nBlock);	Function for sector write
void (* eventHandlerset)(DRV_HANDLE handle, const void * eventHandler, const uintptr_t context);	Function register the event handler with media
SYS_FS_MEDIA_COMMAND_STATUS (* commandStatusGet)(DRV_HANDLE handle, SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE commandHandle);	Function to obtain the command status
void (* Read)(DRV_HANDLE clientHandle, SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE * commandHandle, void * buffer, uint32_t blockStart, uint32_t nBlock);	Function to read certain bytes from the media
uintptr_t (* addressGet)(const DRV_HANDLE hClient);	Function to obtain the address of the media (to be used for NVM only)
DRV_HANDLE (* open)(SYS_MODULE_INDEX index, DRV_IO_INTENT intent);	Function to open the media driver
void (* close)(DRV_HANDLE client);	Function to close the media
void (* tasks)(SYS_MODULE_OBJ obj);	Task function of the media

Description

Media function pointers

This structure contains the definition for functions of media driver, which is registered with the media manager. In future, if any new driver needs to be registered with the media manager (say, to act as a media for file system), the new driver should have implemented all these functions.

Remarks

None.

SYS_FS_MEDIA_MOUNT_DATA Structure

Structure to obtain the device and mount name of media

File

[sys_fs_media_manager.h](#)

C

```

typedef struct {
    const char* mountName;
    const char* devName;
    SYS_FS_MEDIA_TYPE mediaType;
}

```

```

    SYS_FS_FILE_SYSTEM_TYPE fsType;
} SYS_FS_MEDIA_MOUNT_DATA;

```

Members

Members	Description
const char* mountName;	Media Mount Name
const char* devName;	Media Device Name
SYS_FS_MEDIA_TYPE mediaType;	Media Type
SYS_FS_FILE_SYSTEM_TYPE fsType;	File system type on Media

Description

Media Mount Data

This structure is an input for the media manager to auto mount the media when the auto mount feature is enabled.

Remarks

None.

SYS_FS_MEDIA_PROPERTY Enumeration

Contains information of property of a media.

File

[sys_fs_media_manager.h](#)

C

```

typedef enum {
    SYS_FS_MEDIA_SUPPORTS_BYTE_WRITES = SYS_MEDIA_SUPPORTS_BYTE_WRITES,
    SYS_FS_MEDIA_SUPPORTS_READ_ONLY = SYS_MEDIA_SUPPORTS_READ_ONLY,
    SYS_FS_MEDIA_SUPPORTS_ONE_TIME_PROGRAMMING = SYS_MEDIA_SUPPORTS_ONE_TIME_PROGRAMMING,
    SYS_FS_MEDIA_READ_IS_BLOCKING = SYS_MEDIA_READ_IS_BLOCKING,
    SYS_FS_MEDIA_WRITE_IS_BLOCKING = SYS_MEDIA_WRITE_IS_BLOCKING
} SYS_FS_MEDIA_PROPERTY;

```

Members

Members	Description
SYS_FS_MEDIA_SUPPORTS_BYTE_WRITES = SYS_MEDIA_SUPPORTS_BYTE_WRITES	Media supports Byte Write
SYS_FS_MEDIA_SUPPORTS_READ_ONLY = SYS_MEDIA_SUPPORTS_READ_ONLY	Media supports only Read operation
SYS_FS_MEDIA_SUPPORTS_ONE_TIME_PROGRAMMING = SYS_MEDIA_SUPPORTS_ONE_TIME_PROGRAMMING	Media supports OTP (One Time Programming)
SYS_FS_MEDIA_READ_IS_BLOCKING = SYS_MEDIA_READ_IS_BLOCKING	Read in blocking
SYS_FS_MEDIA_WRITE_IS_BLOCKING = SYS_MEDIA_WRITE_IS_BLOCKING	Write is blocking

Description

SYS FS Media Property Structure

This structure contains the information of property of a media device.

Remarks

For a device, if multiple properties are applicable, they can be ORed together and used.

SYS_FS_MEDIA_STATE Enumeration

The enumeration for state of media.

File

[sys_fs_media_manager.h](#)

C

```
typedef enum {
    SYS_FS_MEDIA_STATE_DEREGISTERED = 0,
    SYS_FS_MEDIA_STATE_REGISTERED,
    SYS_FS_MEDIA_CHECK_ATTACH_STATUS,
    SYS_FS_MEDIA_READ_FIRST_SECTOR,
    SYS_FS_MEDIA_ANALYZE_FS
} SYS_FS_MEDIA_STATE;
```

Members

Members	Description
SYS_FS_MEDIA_STATE_DEREGISTERED = 0	Media is de registered with the media manager
SYS_FS_MEDIA_STATE_REGISTERED	Media is registered with the media manager
SYS_FS_MEDIA_CHECK_ATTACH_STATUS	Check the attach/detach status of the Media
SYS_FS_MEDIA_READ_FIRST_SECTOR	Read the first sector of the media
SYS_FS_MEDIA_ANALYZE_FS	Analyze the FS

Description

State of media

The media manager task picks a disk for analysis and takes it through a number of states. This enumeration mentions the state of the media.

Remarks

None.

SYS_FS_MEDIA_STATUS Enumeration

The state of media.

File

[sys_fs_media_manager.h](#)

C

```
typedef enum {
    SYS_FS_MEDIA_DETACHED = SYS_MEDIA_DETACHED,
    SYS_FS_MEDIA_ATTACHED = SYS_MEDIA_ATTACHED
} SYS_FS_MEDIA_STATUS;
```

Members

Members	Description
SYS_FS_MEDIA_DETACHED = SYS_MEDIA_DETACHED	Media is detached
SYS_FS_MEDIA_ATTACHED = SYS_MEDIA_ATTACHED	Media is attached

Description

Status of media

This enumeration states if the media is attached or not

Remarks

None.

SYS_FS_MEDIA_TYPE Enumeration

The enumeration for type of media.

File

[sys_fs_media_manager.h](#)

C

```
typedef enum {
    SYS_FS_MEDIA_TYPE_NVM,
    SYS_FS_MEDIA_TYPE_MSD,
    SYS_FS_MEDIA_TYPE_SD_CARD,
    SYS_FS_MEDIA_TYPE_RAM,
    SYS_FS_MEDIA_TYPE_SPIFLASH
} SYS_FS_MEDIA_TYPE;
```

Members

Members	Description
SYS_FS_MEDIA_TYPE_NVM	Media is of type NVM (internal flash (non volatile) memory)
SYS_FS_MEDIA_TYPE_MSD	Media is of type mass storage device
SYS_FS_MEDIA_TYPE_SD_CARD	Media is of type SD card
SYS_FS_MEDIA_TYPE_RAM	Media is of type RAM
SYS_FS_MEDIA_TYPE_SPIFLASH	Media is of type SPI Flash

Description

Type of media

This enumeration is for the type of media registered with the media manager.

Remarks

None.

SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE Type

Handle identifying block commands of the media.

File

[sys_fs_media_manager.h](#)

C

```
typedef SYS_MEDIA_BLOCK_COMMAND_HANDLE SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE;
```

Description

SYS FS Media 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.

SYS_FS_MEDIA_EVENT_HANDLER Type

Pointer to the Media Event Handler function.

File

[sys_fs_media_manager.h](#)

C

```
typedef SYS_MEDIA_EVENT_HANDLER SYS_FS_MEDIA_EVENT_HANDLER;
```

Returns

None.

Description

Media Event Handler function pointer

This data type defines the required function signature for the media 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.

Remarks

None.

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the media operation requests
context	Value identifying the context of the application that registered the event handling function

SYS_FS_MEDIA_GEOMETRY Type

Contains all the geometrical information of a media device.

File

[sys_fs_media_manager.h](#)

C

```
typedef SYS_MEDIA_GEOMETRY SYS_FS_MEDIA_GEOMETRY;
```

Description

SYS FS Media Device Geometry

This structure contains all the geometrical information of a media device. the structure also has property of the media like media is one time programmable (OTP) or Read Only etc.

Remarks

A memory device can have multiple erase block regions. Sum of all the regions is the total memory size of the device.

SYS_FS_MEDIA_HANDLE Type

Handle identifying the media registered with the media manager.

File[sys_fs_media_manager.h](#)**C**`typedef uintptr_t SYS_FS_MEDIA_HANDLE;`**Description**

SYS FS Media Handle

The media drivers register the media services with the media manager by calling the [SYS_FS_MEDIA_MANAGER_Register](#) function. On successful registration a media handle is returned which can be used by the media driver to deregister the services from the media manager layer.

Remarks

None.

SYS_FS_MEDIA_REGION_GEOMETRY Type

Contains information of a sys media region.

File[sys_fs_media_manager.h](#)**C**`typedef SYS_MEDIA_REGION_GEOMETRY SYS_FS_MEDIA_REGION_GEOMETRY;`**Description**

SYS FS Media Region Geometry Structure

This structure contains the information of a sys media region.

Remarks

A media can have multiple regions. Sum of size of all the regions is the total memory size of the media. Each region is further divided into blocks of identical size.

SYS_FS_VOLUME_PROPERTY Structure

Structure to obtain the property of volume

File[sys_fs_media_manager.h](#)**C**

```
typedef struct _SYS_FS_VOLUME_PROPERTY {
    unsigned int volNumber;
    SYS_FS_FILE_SYSTEM_TYPE fsType;
} SYS_FS_VOLUME_PROPERTY;
```

Members

Members	Description
unsigned int volNumber;	Volume
SYS_FS_FILE_SYSTEM_TYPE fsType;	File system type

Description

Volume property

This structure is passed by sys_fs layer to know the property of a volume. The function

"SYS_FS_MEDIA_MANAGER_VolumePropertyGet" is used for the call.

Remarks

None.

Files

Files

Name	Description
sys_fs.h	Functions and type declarations required to interact with the MPLAB Harmony File System Service.
sys_fs_config_template.h	This is file sys_fs_config_template.h.
sys_fs_media_manager.h	File System Media Manager interface declarations and types.

Description

This section lists the source and header files used by the library.

sys_fs.h

Functions and type declarations required to interact with the MPLAB Harmony File System Service.

Enumerations

	Name	Description
	SYS_FS_ERROR	Lists the various error cases.
	SYS_FS_EVENT	Identifies the possible file system events.
	SYS_FS_FILE_DIR_ATTR	Enumerated data type identifying the various attributes for file/directory.
	SYS_FS_FILE_OPEN_ATTRIBUTES	Lists the various attributes (modes) in which a file can be opened.
	SYS_FS_FILE_SEEK_CONTROL	Lists the various modes of file seek.
	SYS_FS_FILE_SYSTEM_TYPE	Enumerated data type identifying native file systems supported.
	SYS_FS_FORMAT	Specifies the partitioning rule.
	SYS_FS_RESULT	Lists the various results of a file operation.

Functions

	Name	Description
≡◊	SYS_FS_CurrentDriveGet	Gets the current drive
≡◊	SYS_FS_CurrentDriveSet	Sets the drive.
≡◊	SYS_FS_CurrentWorkingDirectoryGet	Gets the current working directory
≡◊	SYS_FS_DirClose	Closes an opened directory.
≡◊	SYS_FS_DirectoryChange	Changes to a the directory specified.
≡◊	SYS_FS_DirectoryMake	Makes a directory.
≡◊	SYS_FS_DirOpen	Open a directory
≡◊	SYS_FS_DirRead	Reads the files and directories of the specified directory.
≡◊	SYS_FS_DirRewind	Rewinds to the beginning of the directory.
≡◊	SYS_FS_DirSearch	Searches for a file or directory.
≡◊	SYS_FS_DriveFormat	Formats a drive.
≡◊	SYS_FS_DriveLabelGet	Gets the drive label.
≡◊	SYS_FS_DriveLabelSet	Sets the drive label
≡◊	SYS_FS_DrivePartition	Partitions a physical drive (media).
≡◊	SYS_FS_DriveSectorGet	Obtains total number of sectors and number of free sectors for the specified drive.

SYS_FS_Error	Returns the last error.
SYS_FS_EventHandlerSet	Allows a client to identify an event handling function for the file system to call back when mount/unmount operation has completed.
SYS_FS_FileCharacterPut	Writes a character to a file.
SYS_FS_FileClose	Closes a file.
SYS_FS_FileDirectoryModeSet	Sets the mode for the file or directory.
SYS_FS_FileDirectoryRemove	Removes a file or directory.
SYS_FS_FileDirectoryRenameMove	Renames or moves a file or directory.
SYS_FS_FileDirectoryTimeSet	Sets or changes the time for a file or directory.
SYS_FS_FileEOF	Checks for end of file.
SYS_FS_FileError	Returns the file specific error.
SYS_FS_FileNameGet	Reads the file name.
SYS_FS_FileOpen	Opens a file.
SYS_FS_FilePrintf	Writes a formatted string into a file.
SYS_FS_FileRead	Read data from the file.
SYS_FS_FileSeek	Moves the file pointer by the requested offset.
SYS_FS_FileSize	Returns the size of the file in bytes.
SYS_FS_FileStat	Gets file status.
SYS_FS_FileStringGet	Reads a string from the file into a buffer.
SYS_FS_FileStringPut	Writes a string to a file.
SYS_FSFileSync	Flushes the cached information when writing to a file.
SYS_FS_FileTell	Obtains the file pointer position.
SYS_FS_FileTestError	Checks for errors in the file.
SYS_FS_FileTruncate	Truncates a file
SYS_FS_FileWrite	Writes data to the file.
SYS_FS_Initialize	Initializes the file system abstraction layer (sys_fs layer).
SYS_FS_Mount	Mounts the file system.
SYS_FS_Tasks	Maintains the File System tasks and functionalities.
SYS_FS_Unmount	Unmounts the file system.

Macros

	Name	Description
	FAT_FS_MAX_LFN	Maximum length of the Long File Name.
	FAT_FS_MAX_SS	Lists the definitions for FAT file system sector size.
	FAT_FS_USE_LFN	Lists the definitions for FAT file system LFN selection.
	SYS_FS_HANDLE_INVALID	Invalid file handle

Structures

	Name	Description
	SYS_FS_FSTAT	File System status
	SYS_FS_FUNCTIONS	SYS FS Function signature structure for native file systems.
	SYS_FS_REGISTRATION_TABLE	The sys_fs layer has to be initialized by passing this structure with suitably initialized members.

Types

	Name	Description
	SYS_FS_EVENT_HANDLER	Pointer to the File system Handler function.
	SYS_FS_HANDLE	This type defines the file handle.

Unions

	Name	Description
	SYS_FS_TIME	The structure to specify the time for a file or directory.

Description

File System Service Library Interface Declarations and Types

This file contains function and type declarations required to interact with the MPLAB Harmony File System Service.

File Name

sys_fs.h

Company

Microchip Technology Inc.

[sys_fs_config_template.h](#)

This is file sys_fs_config_template.h.

[sys_fs_media_manager.h](#)

File System Media Manager interface declarations and types.

Enumerations

	Name	Description
	SYS_FS_MEDIA_BLOCK_EVENT	Identifies the possible events that can result from a request.
	SYS_FS_MEDIA_COMMAND_STATUS	The enumeration for status of buffer
	SYS_FS_MEDIA_PROPERTY	Contains information of property of a media.
	SYS_FS_MEDIA_STATE	The enumeration for state of media.
	SYS_FS_MEDIA_STATUS	The state of media.
	SYS_FS_MEDIA_TYPE	The enumeration for type of media.

Functions

	Name	Description
≡◊	SYS_FS_MEDIA_MANAGER_AddressGet	Gets the starting media address based on a disk number.
≡◊	SYS_FS_MEDIA_MANAGER_CommandStatusGet	Gets the command status.
≡◊	SYS_FS_MEDIA_MANAGER_DeRegister	Function called by a media to deregister itself to the media manager. For static media, (like NVM or SD card), this "deregister function" is never called, since static media never gets deregistered once they are initialized. For dynamic media (like MSD), this register function is called dynamically, once the MSD media is connected.
≡◊	SYS_FS_MEDIA_MANAGER_EventHandlerSet	Register the event handler for Mount/Un-Mount events.
≡◊	SYS_FS_MEDIA_MANAGER_GetMediaGeometry	Gets the media geometry information.
≡◊	SYS_FS_MEDIA_MANAGER_MediaStatusGet	Gets the media status.
≡◊	SYS_FS_MEDIA_MANAGER_Read	Gets data from a specific media address.
≡◊	SYS_FS_MEDIA_MANAGER_Register	Function to register media drivers with the media manager.
≡◊	SYS_FS_MEDIA_MANAGER_RegisterTransferHandler	Register the event handler for data transfer events.
≡◊	SYS_FS_MEDIA_MANAGER_SectorRead	Reads a specified media sector.
≡◊	SYS_FS_MEDIA_MANAGER_SectorWrite	Writes a sector to the specified media.
≡◊	SYS_FS_MEDIA_MANAGER_Tasks	Media manager task function.
≡◊	SYS_FS_MEDIA_MANAGER_TransferTask	Media manager transfer task function.

	SYS_FS_MEDIA_MANAGER_VolumePropertyGet	Gets the volume property.
---	--	---------------------------

Macros

	Name	Description
	SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID	Defines the invalid media block command handle.
	SYS_FS_MEDIA_HANDLE_INVALID	Defines the invalid media handle.

Structures

	Name	Description
	_SYS_FS_VOLUME_PROPERTY	Structure to obtain the property of volume
	SYS_FS_MEDIA_FUNCTIONS	Structure of function pointers for media driver
	SYS_FS_MEDIA_MOUNT_DATA	Structure to obtain the device and mount name of media
	SYS_FS_VOLUME_PROPERTY	Structure to obtain the property of volume

Types

	Name	Description
	SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the media.
	SYS_FS_MEDIA_EVENT_HANDLER	Pointer to the Media Event Handler function.
	SYS_FS_MEDIA_GEOMETRY	Contains all the geometrical information of a media device.
	SYS_FS_MEDIA_HANDLE	Handle identifying the media registered with the media manager.
	SYS_FS_MEDIA_REGION_GEOMETRY	Contains information of a sys media region.

Description

File System Service Media Manager Interface Declarations and Types

This file contains function and type declarations required to interact with the MPLAB Harmony File System Media Manager Framework.

File Name

`sys_fs_media_manager.h`

Company

Microchip Technology Inc.

Interrupt System Service Library

Introduction

This library provides an interface to manage and control interrupt controllers.

Description

This library provides a low-level abstraction of the Interrupt System Service Library that is available on the Microchip family of PIC32 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/PLIB, thereby hiding differences from one microcontroller variant to another.

Using the Library

This topic describes the basic architecture of the Interrupt System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_int.h](#)

The interface to the Interrupt System Service library is defined in the `sys_int.h` header file, which is included by the `sys.h` system service header file. Any C language source (.c) file that uses the Interrupt System Service library should include `sys.h`. Please refer to the [What is MPLAB Harmony?](#) section for how the library interacts with the framework.

Abstraction Model

This library provides an abstraction of the interrupt subsystem that is used by device drivers, middleware libraries and applications to receive and control interrupts in real time.

Description

Interrupt System Service

The interrupt system services provide support for initializing the processor's interrupt controller, managing Interrupt Service Routines (ISRs) and managing interrupts.

Initialization

Each software module (device driver, library, or application) that needs to receive an interrupt must enable that interrupt itself. This is normally done in the module's initialization routine which is called by the `SYS_Initialize` service.

A module that intends to use an interrupt must first register the `Tasks` function that is to be called when the desired source causes an interrupt. Then, it must enable that source, once it is ready to start receiving interrupts.

If the interrupt system service is configured for static usage, the routine that dynamically registers the `Tasks` function will be nulled out by a macro (generating no run-time code) and, instead, the `Tasks` routine must be called statically from the function that implements the raw ISR vector. How this is done is different for each processor family, as explained in the following section.

Interrupt Service Routine (ISR)

Each software module (device driver, library, or application) that needs to receive an interrupt must implement a `Tasks` routine to handle that interrupt. In order for the module to operate in an interrupt-driven mode, the `Tasks` routine must be called from within the appropriate "raw" Interrupt Service Routine (ISR).

How the raw ISR is implemented is highly dependent upon the specific processor being used. Libraries are available that implement raw ISRs for each processor family in a way that allows dynamic registration and deregistration of `Tasks` routines. These libraries maintain tables that associate the `Tasks` routine registered by the `SYS_INIT` service with each interrupt source in the system.

Alternately, in a statically-linked system implementation, the ISR may be implemented by the system designer or integrator (in the configuration-specific `system_interrupt.c` file). Such "static" ISR implementations must identify the source of the interrupt then directly call the appropriate module's `Tasks` routine. This requires knowledge of the modules that have been included in the system and cannot be implemented in advance as a library.



Note: It is also possible, in a highly optimized system (or to support highly resource-restricted parts), to implement the logic of the module's `Tasks` routine directly in the raw ISR. However, this method is not recommended unless absolutely necessary to meet system timing requirements.

Board Support Packages (BSPs)

If the processor is affixed directly to the board, the BSP may also implement any required "raw" ISRs, eliminating the need for the system designer or integrator to implement the ISR(s) himself. Refer to the documentation for the BSP in use for details on what initialization and ISR support it provides. This support is not implemented by the Interrupt System Services library.

How the Library Works

The Interrupt System Service Library can be used by a device driver, middleware layer, or application to provide access to, and control over, interrupts to the processor.

Description

The following diagram describes the major components of the usage model.



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.

Interrupt System Setup

The Interrupt System Service library must be initialized by calling the SYS_INT_Initialize function. If the MPLAB Harmony dynamic initialization service is used, the SYS_INT_Initialize function will be called automatically when the SYS_Initialize function is called. In a statically initialized system, the system designer or integrator must implement the SYS_Initialize function and that function must call SYS_INT_Initialize before initializing any modules that might require use of the interrupt system service. Once the library has been initialized, call the function **SYS_INT_Enable** to enable interrupts to the processor. However, before enabling the generation of interrupts to the processor, each individual module (driver, library, or application) must have a "Tasks" routine to in place (either registered with **SYS_INT_DynamicRegister** or statically linked to the raw ISR) to handle the interrupt before it enables its own interrupt.

Example: Initializing the System Interrupt Library

```
SYS_INT_Initialize();
// Initialize all interrupt-aware software modules
SYS_INT_Enable();
```

Critical Sections

Critical Sections

Critical sections of code are small sections of code that must execute atomically, with no possibility of being interrupted. To support this, the following technique can be used.

Global Interrupt Management provides routines to create a global critical section of code.

Global Critical Section

If no interrupts of any kind can be allowed within a critical section of code, the following routines can be used to ensure this.

- **SYS_INT_Disable**: To start a critical section, all interrupts are disabled with the call of this function
- **SYS_INT_Enable** : To end a critical section, interrupts are enabled from the interrupt controller to the core
- **SYS_INT_IsEnabled**: Status to indicate if whether or not interrupts are currently enabled

Example: Global Critical Section

```
bool flag;
flag = SYS_INT_Disable();
```

```
// Do something critical
```

```
if (flag)
{
    SYS_INT_Enable();
}
```

Source Interrupt Management provides interface routines to create local critical sections.

Local Critical Sections

Normally, it is not necessary to globally disable all possible interrupts. For example, in a driver for a specific device, it is not normally important if an unrelated interrupt occurs in the middle of a critical section of code. However, if the interrupt for the source that the driver manages must not occur within a critical section of code, it can be protected using the following technique.

Example: Local Critical Section

```
bool flag;
// interrupt source enable status before disable is called
flag = SYS_INT_SourceDisable(MY_DRIVER_INTERRUPT_SOURCE);
```

```
// Do something critical
```

```
if (flag)
{
    SYS_INT_SourceEnable(MY_DRIVER_INTERRUPT_SOURCE);
}
```



Note: These methods of protecting critical sections is usually implemented as part of an Operating System Abstraction Layer (OSAL), so it is not normally necessary to use these examples explicitly. Normally, the OSAL will provide single functions or macros that implement this functionality. So, if available, an OSAL method is preferred over implementing the critical section code as shown in the previous examples.

Source Interrupt Management

The driver, middleware, or application's interrupt-handling Tasks routine must do two things at a minimum, in the following order.

1. Remove the cause of the interrupt.
2. Clear the interrupt source by calling the function [SYS_INT_SourceStatusClear](#).

Exactly what actions are necessary to remove the cause of an interrupt is completely dependent on the source of the interrupt. This is normally the main purpose of the driver itself and is beyond the scope of this section. Refer to the documentation for the peripheral being managed.

Warning The cause of the interrupt must be removed before clearing the interrupt source or the interrupt may reoccur immediately after the source is cleared potentially causing an infinite loop. An infinite loop may also occur if the source is not cleared before the interrupt-handler returns.

Example: Handling Interrupts

```
void DRV_MYDEV_Tasks( SYS_MODULE_OBJ object )
{
    // Remove the cause of the interrupt
    //...

    // Clear Interrupt source
    SYS_INT_SourceStatusClear(myIntSourceID);
}
```



Note: The value of `myIntSourceID` is usually either a static or dynamic configuration option. Refer to the documentation for the specific device driver to identify how to define the interrupt source ID.

Testing Interrupt

Sometimes it is necessary to cause an interrupt in software, possibly for testing purposes. To support this, the function [SYS_INT_SourceStatusSet](#) is provided.

Example: Causing an Interrupt in Software

```
SYS_INT_SourceStatusSet(MY_DRIVER_INTERRUPT_SOURCE);
```



Note: This feature is not available for all interrupt sources on all Microchip microcontrollers. Refer to the specific device data sheet to determine whether it is possible for software to set a specific interrupt source.

Configuring the Library

This section provides information on configuring the Interrupt System Service Library.

Description

To use the Interrupt System Service Library, the following must be correctly configured:

- Select the Appropriate Processor
- Initialize the Interrupt System Service
- Configure the Raw ISR Support

Select the Appropriate Processor

The following data types are dependent on the processor selection and are actually defined in the Interrupt Peripheral Library for the specific microcontroller being used.

- [INT_SOURCE](#)
- [INT_PRIORITY](#)
- [INT_SUBPRIORITY](#)

These data types are configured by selecting the appropriate processor in MPLAB X IDE, which adds the "mprocessor" option to the compiler command line to identify the correct processor and processor-specific implementation of the peripheral library to use. Since the Interrupt System Service Library is part of the Microchip Firmware Framework, it will be built with the correct definition of these data types.

Initialize the Interrupt System Service

There are two ways to initialize the interrupt system service, depending on whether you are using a static configuration or a dynamic configuration.

For a Dynamic configuration the constant [SYS_INT_DYNAMIC](#) needs to be defined. This makes the [SYS_INT_DynamicRegister](#) and [SYS_INT_DynamicDeregister](#) functions available. The required driver tasks routines need to be registered using [SYS_INT_DynamicRegister](#) function.

For a Static configuration, the system designer or integrator must implement the [SYS_INT_Initialize](#) function. This function's purpose is to perform any actions necessary to initialize the interrupt subsystem and interrupt controller on the specific processor and system, usually interacting directly with the Interrupt Peripheral Library to accomplish these tasks.

Configure the Raw ISR Support

In some systems, there may only be a single actual (raw) ISR to handle all interrupts. In this sort of system, most of the Interrupt System Service Library may be implemented in software, with only the highest level interrupt being supported by hardware. In other systems, all interrupts may be supported by separate ISRs and vector selection and prioritization will be supported by hardware.

ISRs may be dynamically linked to specific interrupt sources or they may be statically linked at build time. If a dynamic interrupt library is used (by defining the constant [SYS_INT_DYNAMIC](#)), the calls to the [SYS_INT_DynamicRegister](#) function will register a pointer to the given Tasks routine for each registered interrupt source in an internal table. The dynamic library will then determine the source of the interrupt and call the given Tasks routine.

If a static configuration is desired, the "raw" ISR support must be implemented so that it directly calls (using static, build-time linkage) the appropriate module's Tasks routine. This requires the system implementer or integrator to implement the raw ISR, but it reduces the amount of overhead necessary to handle interrupts, reducing both interrupt latency and code size.

Static Configuration

When statically configuring raw ISR support, the system implementer or integrator must directly implement the raw ISRs in an appropriate manner for the selected processor. The raw ISR, must then call the appropriate Tasks routine to properly handle and clear the interrupt source.

Description

A static configuration of the raw ISR support for MPLAB Harmony requires processor-family-specific knowledge. Or, more accurately, it requires compiler-specific knowledge. The following example shows how to implement a raw ISR for the PIC32 family of devices. Refer to the compiler manual for details of how to implement an ISR.

Raw ISR Responsibilities:

- Identify the interrupt source
- Call the appropriate module's Tasks routine

The first thing a raw ISR must do is identify the source of the interrupt. Each interrupt source has its own interrupt "vector". This means that the only time a specific ISR is called is when a specific source has caused an interrupt. Therefore, the raw ISR can assume that every time it is called, its source has caused an interrupt. Once the raw ISR has identified the interrupt source, it must call the appropriate module's Tasks routine to service and clear the interrupt.

Example: PIC32 Timer1 Raw ISR

```
void __ISR ( _TIMER_1_VECTOR ) _InterruptHandler_TMR_1_stub( void )
{
    /* Call the timer driver's "Tasks" routine */
    DRV_TMR_Tasks ( gTMRObject );
}
```

In the example, gTMRObject holds the return value from the DRV_TMR_Initialize function.

The [SYS_INT_DynamicRegister](#) and [SYS_INT_DynamicDeregister](#) functions are macro switched to compile away to nothing if a static configuration is chosen.

Dynamic Configuration

When dynamically configuring raw ISR support, the system implementer or integrator must register each interrupt-driven driver or module's Tasks routine with the dynamic system interrupt service for the appropriate interrupt source. The dynamic SYS INT service will then ensure that the appropriate Tasks routine is called when an interrupt occurs.

Description

When using the dynamic system interrupt (SYS INT) service, it is not necessary to implement raw ISRs for interrupt-driven modules. The processor-family-specific, dynamic SYS INT implementation provided with MPLAB Harmony implements the raw ISRs so the system developer or integrator does not have to. Instead, the system developer must register the module's Tasks" routine using the [SYS_ModuleRegister](#) function after registering the module in the module registration routine (described in the SYS INIT documentation). The following example shows how a module must register its ISR Tasks routine.

Example: Dynamic Registration of an Interrupt-Driven Module

```
// Register the TMR driver's "Tasks" routine with the SYS INT service
SYS_INT_DynamicRegister(object, DRV_TMR_Tasks, PLIB_INT_SOURCE_TIMER_1);
```

The module init routine must register the module's Tasks routine with the SYS INT service instead of the SYS TASKS service. To do this, it calls the [SYS_INT_DynamicRegister](#) function, passing in the object (the same object handle returned by the module's initialization routine), along with a pointer to the module's Tasks routine and the interrupt source with which it will be associated.



Note: Dynamic interrupt registration functionality is currently not supported in the Interrupt System Service.

Building the Library

This section lists the files that are available in the Interrupt System Service 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/system/int.

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
sys_int.h	Interrupt System Service Library API header file.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

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/sys_int_pic32.c	Interrupt System Service Library implementation.

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 Interrupt System Service does not depend on any other modules.

Library Interface

a) Global Interrupt Management Functions

	Name	Description
≡	SYS_INT_Disable	Disable Global Interrupt
≡	SYS_INT_Enable	Enable Global Interrupt
≡	SYS_INT_IsEnabled	Returns the interrupt controller's global enable/disable status.
≡	SYS_INT_Restore	Restores the interrupt controller to the state specified in the parameter.

b) Interrupt Source Management Functions

	Name	Description
≡	SYS_INT_SourceDisable	Disables the specified interrupt source.
≡	SYS_INT_SourceEnable	Enables the specified interrupt source.
≡	SYS_INT_SourceStatusClear	Clears the pending status of the specified interrupt source.
≡	SYS_INT_SourceStatusGet	Gets the pending status of the specified interrupt source.
≡	SYS_INT_SourceStatusSet	Sets the pending status of the specified interrupt source manually.
≡	SYS_INT_SourcesEnabled	Gets the enable/disable status of the specified interrupt source.

	SYS_INT_DynamicRegister@INT_SOURCE@SYS_INT_TASKS_POINTER@SYS_MODULE_OBJ	<p>The System Interrupt library must be initialized by calling the <code>SYS_INT_Initialize</code> routine. This is normally done in the <code>"SYS_Initialize"</code> routine before any interrupt support is used. If the dynamic interrupt system service is not used, the <code>"SYS_Initialize"</code> routine must be implemented by the system designer or integrator as required by the system design and is not implemented by the System Interrupt library. Once the library has been initialized, call the function SYS_INT_Enable to enable interrupts to the processor. However, before enabling the generation of interrupts to the processor, each individual module (driver, library, or application) must have a <code>"Tasks"</code> routine... more</p>
--	---	--

c) Data Types and Constants

	Name	Description
	INT_SOURCE	Defines the data type for all the interrupt sources associated with the interrupt controller of the device.

Description

This section describes the APIs of the Interrupt System Service Library.

Refer to each section for a detailed description.

a) Global Interrupt Management Functions

SYS_INT_Disable Function

Disable Global Interrupt

File

[sys_int.h](#)

C

```
bool SYS_INT_Disable();
```

Returns

This function disables the global interrupt and return the state of global interrupt prior to disabling it. The state information will be used to restore the global interrupt to the original state after the critical section.

- true - Global Interrupt is enabled
- false - Global Interrupt is disabled

Description

This function disables global interrupt and returns the state of the global interrupt prior to disabling it. When global interrupt is disabled, only NMI and HardFault exceptions are allowed. This may be used to disable global interrupt during critical section and restore the global interrupt state after the critical section.

Remarks

Returned status can be passed to [SYS_INT_Restore](#) to restore the previous global interrupt status (whether it was enabled or disabled).

Preconditions

None.

Example

```
bool interruptState;

// Save global interrupt state and disable interrupt
interruptState = SYS_INT_Disable();

// Critical Section

// Restore interrupt state
SYS_INT_Restore(interruptState)
```

Function

```
bool SYS_INT_Disable( void )
```

SYS_INT_Enable Function

Enable Global Interrupt

File

[sys_int.h](#)

C

```
void SYS_INT_Enable();
```

Returns

None.

Description

This function enables global interrupt.

Remarks

None.

Preconditions

None

Example

```
SYS_INT_Enable();
```

Function

```
void SYS_INT_Enable( void )
```

SYS_INT_IsEnabled Function

Returns the interrupt controller's global enable/disable status.

File

[sys_int.h](#)

C

```
bool SYS_INT_IsEnabled();
```

Returns

- true - Global Interrupt is enabled.
- false - Global Interrupt is disabled.

Description

This function returns global interrupt enable status.

Remarks

None.

Preconditions

None.

Example

```
if(true == SYS_INT_IsEnabled())
{
    // Global Interrupt is enabled
}
```

Function

```
bool SYS_INT_IsEnabled( void )
```

SYS_INT_Restore Function

Restores the interrupt controller to the state specified in the parameter.

File[sys_int.h](#)**C**

```
void SYS_INT_Restore(bool state);
```

Returns

None.

Description

This function restores the interrupt controller to the state specified in the parameters.

Remarks

None.

Preconditions

`SYS_INT_Disable` must have been called to get previous state of the global interrupt.

Example

```
bool interruptState;

// Save global interrupt state and disable interrupt
interruptState = SYS_INT_Disable();

// Critical Section

// Restore interrupt state
SYS_INT_Restore(interruptState)
```

Function

```
void SYS_INT_Restore( bool state )
```

b) Interrupt Source Management Functions**SYS_INT_SourceDisable Function**

Disables the specified interrupt source.

File[sys_int.h](#)**C**

```
bool SYS_INT_SourceDisable( INT_SOURCE source );
```

Returns

- true - Interrupt line was enabled.
- false - Interrupt line was disabled.

Description

This function returns the current interrupt enable/disable status and disables the specified interrupt source/line at the interrupt controller level.

Remarks

If the corresponding module level interrupts are enabled and triggered, the triggers will be ignored at interrupt controller. For example, USART0 doesn't cause interrupt unless both the interrupt controller source/line and USART0 module level interrupt for

TX or RX or Error are enabled.

Preconditions

None.

Example

```
bool usart0Int = false;
usart0Int = SYS_INT_SourceDisable(USART0_IRQn);
```

Parameters

Parameters	Description
source	Interrupt source/line available at interrupt controller.

Function

bool SYS_INT_SourceDisable(**INT_SOURCE** source)

SYS_INT_SourceEnable Function

Enables the specified interrupt source.

File

[sys_int.h](#)

C

```
void SYS_INT_SourceEnable(INT_SOURCE source);
```

Returns

None.

Description

This function enables the specified interrupt source/line at the interrupt controller level.

Remarks

The corresponding module level interrupts must be enabled to trigger the specified interrupt source/line. For example, USART0 interrupt enable at interrupt controller level is not triggered unless USART0 module level interrupt for TXRDY or RXRDY or Error interrupts are not enabled.

Preconditions

None.

Example

```
SYS_INT_SourceEnable(USART0_IRQn);
```

Parameters

Parameters	Description
source	Interrupt source/line available at interrupt controller.

Function

void SYS_INT_SourceEnable(**INT_SOURCE** source)

SYS_INT_SourceStatusClear Function

Clears the pending status of the specified interrupt source.

File[sys_int.h](#)**C**`void SYS_INT_SourceStatusClear(INT_SOURCE source);`**Returns**

None.

Description

This function clears the pending status of the specified interrupt source at the interrupt controller level. It is ignored if the interrupt condition has already been cleared in hardware.

Remarks

None.

Preconditions

None.

Example

```
//Clear a pending interrupt.
SYS_INT_SourceStatusClear(USART0_IRQn);
```

Parameters

Parameters	Description
source	Interrupt source/line available at interrupt controller.

Function`void SYS_INT_SourceStatusClear(INT_SOURCE source)`**SYS_INT_SourceStatusGet Function**

Gets the pending status of the specified interrupt source.

File[sys_int.h](#)**C**`bool SYS_INT_SourceStatusGet(INT_SOURCE source);`**Returns**

- true - Interrupt status is pending.
- false - Interrupt status is not pending.

Description

This function returns the pending status of the specified interrupt source at the interrupt controller level.

Remarks

Interrupt pending status may get cleared automatically once the corresponding interrupt vector executes on some devices.

Preconditions

None.

Example

```
bool usart0IntStatus = SYS_INT_SourceStatusGet(USART0_IRQn);
```

Parameters

Parameters	Description
source	Interrupt source/line available at interrupt controller.

Function

```
bool SYS_INT_SourceStatusGet( INT_SOURCE source )
```

SYS_INT_SourceStatusSet Function

Sets the pending status of the specified interrupt source manually.

File

[sys_int.h](#)

C

```
void SYS_INT_SourceStatusSet(INT_SOURCE source);
```

Returns

None.

Description

This function manually sets the pending status of the specified interrupt source at the interrupt controller level. This triggers interrupt controller for a specified source even though the interrupt condition has not met at hardware.

Remarks

This feature may not be supported by some devices.

Preconditions

None.

Example

```
//Trigger USART0 ISR handler manually
SYS_INT_SourceStatusSet(USART0_IRQn);
```

Parameters

Parameters	Description
source	Interrupt source/line available at interrupt controller.

Function

```
void SYS_INT_SourceStatusSet( INT_SOURCE source )
```

SYS_INT_SourceIsEnabled Function

Gets the enable/disable status of the specified interrupt source.

File

[sys_int.h](#)

C

```
bool SYS_INT_SourceIsEnabled(INT_SOURCE source);
```

Returns

- true - Interrupt line is enabled.
- false - Interrupt line is disabled.

Description

This function returns the enable/disable status of the specified interrupt source/line at the interrupt controller level.

Remarks

Unlike [SYS_INT_Disable](#), this function just returns the status and doesn't disable the interrupt line.

Preconditions

None.

Example

```
bool usart0Int = false;
usart0Int = SYS_INT_SourceIsEnabled(USART0_IRQn);
```

Parameters

Parameters	Description
source	Interrupt source/line available at interrupt controller.

Function

```
bool SYS_INT_SourceIsEnabled( INT_SOURCE source )
```

SYS_INT_DynamicRegister@INT_SOURCE@SYS_INT_TASKS_POINTER@SYS_MODULE_OBJ

The System Interrupt library must be initialized by calling the [SYS_INT_Initialize](#) routine. This is normally done in the "SYS_Initialize" routine before any interrupt support is used.

If the dynamic interrupt system service is not used, the "SYS_Initialize" routine must be implemented by the system designer or integrator as required by the system design and is not implemented by the System Interrupt library.

Once the library has been initialized, call the function [SYS_INT_Enable](#) to enable interrupts to the processor. However, before enabling the generation of interrupts to the processor, each individual module (driver, library, or application) must have a "Tasks" routine to in place (either registered with [SYS_INT_DynamicRegister](#) or statically linked to the raw ISR) to handle the interrupt before it enables its own interrupt.

Example: Initializing the System Interrupt Library

```
// Initialize the interrupt system.
SYS_INT_Initialize();
// Initialize all interrupt-aware software modules
SYS_INT_Enable();
```

c) Data Types and Constants

INT_SOURCE Type

Defines the data type for all the interrupt sources associated with the interrupt controller of the device.

File

[sys_int.h](#)

C

```
typedef IRQn_Type INT_SOURCE;
```

Description

Interrupt Sources

This data type can be used with interface functions to enable, disable, set, clear and to get status of any particular interrupt source.

Remarks

This data type is defined using the CMSIS data type that defines the interrupt sources set available.

Files

Files

Name	Description
sys_int.h	Interrupt system service library interface.

Description

This section lists the source and header files used by the library.

sys_int.h

Interrupt system service library interface.

Functions

	Name	Description
≡	SYS_INT_Disable	Disable Global Interrupt
≡	SYS_INT_Enable	Enable Global Interrupt
≡	SYS_INT_IsEnabled	Returns the interrupt controller's global enable/disable status.
≡	SYS_INT_Restore	Restores the interrupt controller to the state specified in the parameter.
≡	SYS_INT_SourceDisable	Disables the specified interrupt source.
≡	SYS_INT_SourceEnable	Enables the specified interrupt source.
≡	SYS_INT_SourceIsEnabled	Gets the enable/disable status of the specified interrupt source.
≡	SYS_INT_SourceStatusClear	Clears the pending status of the specified interrupt source.
≡	SYS_INT_SourceStatusGet	Gets the pending status of the specified interrupt source.
≡	SYS_INT_SourceStatusSet	Sets the pending status of the specified interrupt source manually.

Types

	Name	Description
	INT_SOURCE	Defines the data type for all the interrupt sources associated with the interrupt controller of the device.

Description

Interrupt System Service Library Interface Header File

This file defines the interface to the interrupt system service library. This library provides access to and control of the interrupt controller.

Remarks

Interrupt controller initialize will be done from within the MCC.

File Name

sys_int.h

Company

Microchip Technology Inc.

Ports System Service Library

Introduction

This library provides an interface to manage and control general purpose input or output ports on the Microchip families of microcontrollers.

Description

Ports System service abstract different general purpose input output (GPIO) modules present on Microchip family of devices. it provides a common interface to use GPIO across all the 32 bit Microchip devices.

Using the Library

This topic describes the basic architecture of the Ports System Service Library and provides information and examples on its use.

Abstraction Model

Configuring the Library

There is no configuration needed to use PORT System Service.

Library Interface

This section describes the APIs of the Ports System Service Library.

Refer to each section for a detailed description.

Data Types and Constants

Files

Files

Name	Description
sys_ports.h	This is file sys_ports.h.

Description

This section lists the source and header files used by the library.

sys_ports.h

This is file sys_ports.h.

System Services Libraries Overview

This section provides an overview to MPLAB Harmony system services and information that is common to all services.

Description

Introduction

Describes system services provided by MPLAB Harmony.

Description

MPLAB Harmony provides system service libraries to support common functionality and manage resources that are shared by multiple drivers, libraries, and other modules.

A system service encapsulates code that manages a shared resource or implements a common capability in a single location so that it does not need to be replicated by individual drivers and libraries. This feature eliminates duplicated code and creates consistency across all modules, and also helps to eliminate potential conflicts and complex configuration issues and runtime interactions, resulting in a smaller and simpler overall solution.

System services may directly manage one or more peripherals or core processor capabilities by utilizing peripheral libraries, special function registers, special CPU instructions, or coprocessor registers. Some system services may utilize drivers, other system services, or even entire middleware stacks to share or emulate a common resource.

System services may be implemented statically (possibly generated by the MPLAB Harmony Configurator (MHC)) or dynamically to support multiple channels and instances like a driver. However, system services will not normally provide common *Open* and *Close* functions like a device driver.

In general, the distinguishing feature of a system service is that it implements a common capability that would otherwise "cut horizontally" across multiple modules in a system, causing interoperability and compatibility conflicts if the capability were implemented by other libraries.

System service functions use the following naming convention:

SYS_<module-abbreviation>_[<feature-short-name>]<operation>

Where,

- `SYS_` indicates that this is a system service function
- `<module-abbreviation>` is the abbreviated name of the system service module to which this function belongs
- `[<feature-short-name>]` is an optional short (or shortened) name that identifies the feature of the associated system service module to which this function refers. The feature short name will appear in the name of all functions that interact with or provide access to that particular feature.
- `<operation>` is a verb that identifies the action taken by the function

For example, `SYS_TMR_AlarmSet`, where:

- `<module-abbreviation>` = `TMR`, which indicates that this is a Timer System Service function
- `<feature-short-name>` = `Alarm`, which indicates that this function controls the alarm feature of the Timer System Service
- `<operation>` = `Set`, which indicates that this function sets the value of the alarm feature of the Timer System Service, as indicated by the function's parameters (not shown above).

System State Machine

Describes the MPLAB Harmony main function and system-wide state machine.

Description

In its most basic configuration, a MPLAB Harmony system operates in a single polled *superloop* that is implemented in the project's *main* function (in the `<project-name>/firmware/src/main.c` file) that is generated by the MHC, as shown in the following example. The *main* function calls two system-wide state machine functions, `SYS_Initialize` and `SYS_Tasks` that are also generated by the MHC to initialize and run the system.

MPLAB Harmony "main" Function

```
MAIN_RETURN main ( void )
{
    SYS_Initialize(NULL);
```

```

while(true)
{
    SYS_Tasks();
}

return(EXIT_FAILURE);
}

```

The `SYS_Initialize` function calls the initialization functions for all library and application modules that are used in the system to place them in their initial states. Each module's initialization function must prepare the module so it is safe to call its `Tasks` function(s). The `SYS_Initialize` function's implementation (and necessary support code) is generated by the MHC in the `<project-name>/firmware/src/system_config/<configuration-name>/system_init.c` file.

After initializing all modules, the `main` function contains the system-wide super loop that executes continuously until the system is powered off or reset. Inside the super loop, the `main` function calls the `SYS_Tasks` function. This function in turn calls the `Tasks` functions for any library or application modules whose state machines operate in a polled manner in the system. Of course, those state machines must first have been initialized by a prior call to their initialization functions by the `SYS_Initialize` function. The `SYS_Task` function's implementation is also generated by the MHC, but it is contained in the `<project-name>/firmware/src/system_config/<configuration-name>/system_tasks.c` file.

The polled super loop configuration described above is the most basic configuration of a MPLAB Harmony project. However, a single MPLAB Harmony project can have multiple configurations and different configurations may change modules so that they operate either polled or interrupt driven or in an RTOS thread. However, regardless of the configuration selected, the `main` function does not normally change.

MPLAB Harmony Documentation Volumes

For further information on MPLAB Harmony configurations and execution models, refer to the following MPLAB Harmony documentation.

- Volume I: Getting Started with MPLAB Harmony
- Volume III: MPLAB Harmony Configurator (MHC)
- Volume IV: MPLAB Harmony Development



Note: Although the MPLAB Harmony `main` function and system state machine functions (and other system configuration code) are generated as part of a MPLAB Harmony project for your convenience, these files are unique to your project and can be modified or implemented by other means if so desired, or even removed from the project to create a binary library (.a file) containing all configured MPLAB Harmony libraries. Refer to the "MPLAB® X IDE User's Guide" (DS50002027) and the "MPLAB® XC32 C/C++ Compiler User's Guide" (DS50001686) for information on creating library projects (both documents are available for download from the Microchip website: www.microchip.com).

MPLAB Harmony Module System Interface

Describes the MPLAB Harmony module system interface and provides usage examples.

Description

To support the system-wide state machine (see [System State Machine](#)), an MPLAB Harmony module must provide an initialization function and a `Tasks` function. In addition, a MPLAB Harmony module may (optionally) provide deinitialization, reinitialization, and status functions. This set of functions is considered the module's "system interface". The system state machine, system scheduler, or any other system-management code uses a module's system interface to initialize, run, and otherwise control the execution of a module in the MPLAB Harmony system.

To define the calling signature of these system interface functions, the `sys_module.h` header defines function pointer data types for each. These data types could be used to develop a dynamic system with capabilities beyond the basic system state machine, such as dynamic task registration, power management, advanced schedulers, or even your own operating system.

The following examples show how the system interface could be used to create a simple dynamic polled system tasks scheduler.

Example Dynamic System Data Structures

```

typedef struct _system_module_interface
{
    SYS_MODULE_INITIALIZE_ROUTINE    initialize;
    SYS_MODULE_TASKS_ROUTINE        tasks;
    SYS_MODULE_REINITIALIZE_ROUTINE reinitialize;
    SYS_MODULE_DEINITIALIZE_ROUTINE deinitialize;
}

```

```

SYS_MODULE_STATUS_ROUTINE      status;

} SYSTEM_MODULE_INTERFACE;

typedef struct _system_module_data
{
    SYSTEM_MODULE_INTERFACE function;
    SYS_MODULE_INDEX      index;
    SYS_MODULE_INIT       *initData;
    SYS_MODULE_OBJ        obj;
    SYS_STATUS            status;
    uint8_t               powerState;

} SYSTEM_MODULE_DATA;

```

```
SYSTEM_MODULE_DATA gModules[CONFIG_NUMBER_OF_MODULES];
```

In the previous example code, the SYSTEM_MODULE_INTERFACE structure contains pointers to all of a module's system interface functions. This structure could be filled in with pointers to a module's system interface functions and a pointer to the structure passed into a dynamic module registration function, along with the module index number and a pointer to any initialization data required by the module. The following example shows how this dynamic registration function might appear.

Example Dynamic Module Registration Function

```

bool SYS_ModuleRegister ( SYSTEM_MODULE_DATA *module,
                           SYS_MODULE_INDEX      index,
                           SYS_MODULE_INIT       *initData )
{
    SYSTEM_MODULE_DATA  module;
    int                i;
    bool               success = false;

    for (i=0; i < CONFIG_NUMBER_OF_MODULES; i++)
    {
        if (gModules[i].function.initialize != NULL)
        {
            module = &gModules[i];
        }
    }

    if (i < CONFIG_NUMBER_OF_MODULES)
    {
        module->function.initialize      = module->initialize;
        module->function.tasks          = module->tasks;
        module->function.reinitialize   = module->reinitialize;
        module->function.deinitialize   = module->deinitialize;
        module->function.status        = module->status;

        module->index      = index;
        module->initData   = initData;
        module->obj       = SYS_MODULE_OBJ_INVALID;
        module->status     = SYS_STATUS_UNINITIALIZED;
        module->powerState = SYS_MODULE_POWER_RUN_FULL;

        success = true;
    }

    return success;
}

```

The SYS_ModuleRegister function could then scan a system-global array (gModules, from the previous system data structures example) to find an empty SYSTEM_MODULE_DATA structure (using the initialization function pointer as a "flag" to indicate if the structure is in use or not), copy the newly registered module's interface and other data into the structure, and initialize the other members of the structure.

Once all modules to be used have been similarly registered, the entire system could be initialized when desired by calling a SYS_InitializeDynamic function implementation, similar to the following example.

Example Dynamic System Initialization Function

```
void SYS_InitializeDynamic ( void *data )
{
    SYSTEM_MODULE_DATA  module = (SYSTEM_MODULE_DATA *)data;
    int                 i;

    for (i=0; i < CONFIG_NUMBER_OF_MODULES; i++)
    {
        if (module->function.initialize != NULL)
        {
            module->obj = module->function.initialize(module->index,
                                              module->initData);

            module->status      = SYS_STATUS_BUSY;
            module->powerState  = SYS_MODULE_POWER_RUN_FULL;

            if (module->obj == SYS_MODULE_OBJ_INVALID)
            {
                module->function.initialize = NULL;
            }
        }
    }
}
```

The previous `SYS_InitializeDynamic` example function iterates through the global array of module data structures, calling the initialization functions for any modules that have been registered and skipping over any structures that have no module registered (again using the required initialization function pointer as a sort of flag). If a module's initialization function successfully initializes the module it returns a valid module object handle, which the dynamic system initialization function captures in the module's data structure so it can call the module's other system interface routines. If the module is not successfully initialized, the object handle reported will be invalid (`SYS_MODULE_OBJECT_INVALID`) and the module is deregistered by nulling the pointer to the initialization function.

Once the system has been initialized, a dynamic tasks function like the following example would also iterate through the module data array and call the tasks functions for each module registered.

Example Dynamic System Tasks Function

```
void SYS_TasksDynamic ( void *data )
{
    SYSTEM_MODULE_DATA  module = (SYSTEM_MODULE_DATA *)data;
    int                 i;

    for (i=0; i < CONFIG_NUMBER_OF_MODULES; i++)
    {
        if (module->function.initialize != NULL &&
            module->function.tasks       != NULL )
        {
            if (module->status(module->obj) >= SYS_STATUS_UNINITIALIZED)
            {
                module->function.tasks(module->obj);
            }
            else
            {
                module->function.deinitialize(module->obj);
                module->function.initialize = NULL;
            }
        }
    }
}
```

After calling a module's `Tasks` function, the previous example checks the module's status by calling its `Status` function. If the module reports an error status (any status less than `SYS_STATUS_UNINITIALIZED`), the module is deinitialized by calling its `Deinitialize` function and deregistered by nulling out the `Initialize` function pointer.

Please note that the example code provided is for informational purposes only, and is used to describe the purpose and usage of the MPLAB Harmony module system interface. A real dynamic tasks scheduler would need to deal with additional complexities. For example, MPLAB Harmony modules may have zero or more `Tasks` functions (only the initialization function is absolutely required). Therefore, a single tasks-function pointer would not be sufficient. The previous example demonstrates usage of the system status and power state data, but it does not actually update or manage these items. Also, this example does not

demonstrate the usage of a module's reinitialization function. This function allows a module to provide a way to change its initial parameters while the module is active (after it has been initialized) without disrupting active clients of the module. *However, at the time of this writing, most MPLAB Harmony modules do not implement this function, so this capability is not usually available.*

Using the SYS_ASSERT Macro

Describes the purpose and usage of the system assertion macro.

Description

The **SYS_ASSERT** macro is a testing and debugging tool used throughout MPLAB Harmony libraries to verify (or assert) critical assumptions before any action is taken on them. However, it is not usually desirable to have these tests in production code because system assertion failures are normally fatal events that stop current execution, either hanging or resetting the system because it is not safe to continue. Also, even if the tests pass, they would add significant overhead, affecting code size and execution time. To avoid these issues, the default implementation of this macro is empty to eliminate the assertion tests and failure messages from the build, as shown in the following example.

Default SYS_ASSERT Macro Definition

```
#define SYS_ASSERT(test,message)
```

Conversely, when developing, debugging, and testing a project, having **SYS_ASSERT** statements in the code (particularly in library code) can be very helpful. This macro is used to check key parameter values and intermediate results and provide messages that explain the consequences that occur when they don't match what was expected when the library was implemented. A **SYS_ASSERT** failure during testing may save significant time spent debugging a MPLAB Harmony library only to find that a configuration setting or value passed into a library was incorrect or unexpected.

The **SYS_ASSERT** macro provides a convenient way to obtain the desired behavior when an assertion fails. In production code, simply accept the default definition that disposes of the assertion test code and the failure message. If things go wrong during testing, either enable a predefined implementation of this macro or define it in any way that is convenient for the current application.

One particularly useful definition of the **SYS_ASSERT** macro is to execute a hard-coded breakpoint instruction if the assertion fails, as shown in the following example.

Example SYS_ASSERT Macro Breakpoint Definition

```
#define SYS_ASSERT(test, message) \
    do{ if(!(test)) SYS_DEBUG_Breakpoint(); }while(false)
```

When using the previous definition, if the assertion test fails, the processor will reach a hard-coded breakpoint and execution will stop in the debugger at the **SYS_ASSERT** call. This behavior identifies the point in the source code where the assertion call failed. Then, the assertion call conveniently provides an explanation of what the failure means in the associated message string. A system assertion failure is a catastrophic failure, so the do...while loop hangs the system to prevent execution of code under an invalid assumption that may cause incorrect behavior or even damage to the system. It is also a way to guarantee that the **if** statement is correctly interpreted by the compiler and not accidentally associated with a following **else** statement.

Another useful implementation of the **SYS_ASSERT** macro might use the Debug System Service to display the message, as shown in the following example.

Example SYS_ASSERT Macro Definition Using Debug Message Service

```
#define SYS_ASSERT(test, message) \
    do{ if(!(test)){ \
        SYS_DEBUG_Message( (message) ); \
        SYS_DEBUG_Message( "\r\n" ); \
        while(1); } \
    }while(false)
```

The previous definition will display the assertion failure message using whatever method the **SYS_DEBUG** service is configured to use. However, it has the drawback that it does not hang the system (because the message service may need to continue running), so incorrect or unsafe behavior may result.

It is also possible to combine these two example definitions, which is particularly useful if the debug message service is configured to use the debugger's output window, or to create any another definition that is more useful for a given situation.

Obtaining System Version Information

Describes the purpose and usage of the system version functions.

Description

It is possible to programmatically obtain the version information for the release of MPLAB Harmony using the `SYS_Version` group of functions. For example, once the Debug System Service has been initialized and is running, the following code would retrieve the current version number and display it on the configured debug console if it were lower than a given value.

Example Version Function Usage

```
#define REQUIRED_VERSION 10700

if (SYS_VersionGet() < REQUIRED_VERSION)
{
    SYS_DEBUG_MESSAGE( SYS_ERROR_WARNING, "Version %s is too low",
                       SYS_VersionStrGet() );
}
```

In the previous example, the `SYS_VersionGet` function is used to get the numeric representation of the version number so that it can be easily compared to a known value and the `SYS_VersionStrGet` is used to get a string representation of it for displaying on the debug console.

Library Interface

Data Types and Constants

	Name	Description
	<code>SYS_MEDIA_GEOMETRY_TABLE_ERASE_ENTRY</code>	Erase Region Geometry Table Index Numbers
	<code>SYS_MEDIA_GEOMETRY_TABLE_READ_ENTRY</code>	Read Region Geometry Table Index Numbers
	<code>SYS_MEDIA_GEOMETRY_TABLE_WRITE_ENTRY</code>	Write Region Geometry Table Index Numbers
	<code>SYS_MEDIA_GEOMETRY</code>	Contains all the geometrical information of a media device.
	<code>SYS_MEDIA_REGION_GEOMETRY</code>	Contains information of a sys media region.
	<code>SYS_MEDIA_PROPERTY</code>	Contains information of property of a media.
	<code>SYS_MEDIA_BLOCK_COMMAND_HANDLE</code>	Handle identifying block commands of the media.
	<code>SYS_MEDIA_BLOCK_EVENT</code>	Identifies the possible events that can result from a request.
	<code>SYS_MEDIA_COMMAND_STATUS</code>	The enumeration for status of buffer
	<code>SYS_MEDIA_EVENT_HANDLER</code>	Pointer to the Media Event Handler function.
	<code>SYS_MEDIA_STATUS</code>	The state of media.

Version Functions

	Name	Description
	<code>SYS_VersionGet</code>	Gets SYS_COMMON version in numerical format.
	<code>SYS_VersionStrGet</code>	Gets SYS_COMMON version in string format.

Description

This section describes the APIs of the System Service Library.

Refer to each section for a detailed description.

Main Function Support

System Assert

System State Machine

Version Functions

SYS_VersionGet Macro

Gets SYS_COMMON version in numerical format.

File

[system_common.h](#)

C

```
#define SYS_VersionGet( void ) SYS_VERSION
```

Returns

Current driver version in numerical format.

Description

This routine gets the SYS_COMMON version. The version is encoded as major * 10000 + minor * 100 + patch. The string version can be obtained using [SYS_VersionStrGet\(\)](#)

Function

SYS_VersionGet(void)

SYS_VersionStrGet Macro

Gets SYS_COMMON version in string format.

File

[system_common.h](#)

C

```
#define SYS_VersionStrGet( void ) SYS_VERSION_STR
```

Returns

Current SYS_COMMON version in the string format.

Description

Macro: char * SYS_VersionStrGet (void)

This routine gets the SYS_COMMON version in string format. The version is returned as major.minor.path[type], where type is optional. The numerical version can be obtained using [SYS_VersionGet\(\)](#)

Remarks

None.

Data Types and Constants

SYS_MEDIA_GEOMETRY_TABLE_ERASE_ENTRY Macro

File

[system_media.h](#)

C

```
#define SYS_MEDIA_GEOMETRY_TABLE_ERASE_ENTRY ( 2 )
```

Description

Erase Region Geometry Table Index Numbers

SYS_MEDIA_GEOMETRY_TABLE_READ_ENTRY Macro

File

[system_media.h](#)

C

```
#define SYS_MEDIA_GEOMETRY_TABLE_READ_ENTRY ( 0 )
```

Description

Read Region Geometry Table Index Numbers

SYS_MEDIA_GEOMETRY_TABLE_WRITE_ENTRY Macro

File

[system_media.h](#)

C

```
#define SYS_MEDIA_GEOMETRY_TABLE_WRITE_ENTRY ( 1 )
```

Description

Write Region Geometry Table Index Numbers

SYS_MEDIA_GEOMETRY Structure

Contains all the geometrical information of a media device.

File

[system_media.h](#)

C

```
typedef struct {
    SYS_MEDIA_PROPERTY mediaProperty;
    uint32_t numReadRegions;
    uint32_t numWriteRegions;
    uint32_t numEraseRegions;
    SYS_MEDIA_REGION_GEOMETRY * geometryTable;
} SYS_MEDIA_GEOMETRY;
```

Members

Members	Description
SYS_MEDIA_PROPERTY mediaProperty;	Properties of a Media. For a device, if multiple properties are applicable, they can be ORed
uint32_t numReadRegions;	Number of Read Regions
uint32_t numWriteRegions;	Number of Write Regions
uint32_t numEraseRegions;	Number of Erase Regions
SYS_MEDIA_REGION_GEOMETRY * geometryTable;	Pointer to the table containing the geometry information

Description

SYS Media Device Geometry

This structure contains all the geometrical information of a media device. the structure also has property of the media like media is one time programmable (OTP) or Read Only etc.

Remarks

A memory device can have multiple erase block regions. Sum of all the regions is the total memory size of the device.

SYS_MEDIA_REGION_GEOMETRY Structure

Contains information of a sys media region.

File

[system_media.h](#)

C

```
typedef struct {
    uint32_t blockSize;
    uint32_t numBlocks;
} SYS_MEDIA_REGION_GEOMETRY;
```

Members

Members	Description
uint32_t blockSize;	Size of a each block in Bytes
uint32_t numBlocks;	Number of Blocks of identical size within the Region

Description

SYS Media Region Geometry Structure

This structure contains the information of a sys media region.

Remarks

A media can have multiple regions. Sum of size of all the regions is the total memory size of the media. Each region is further divided into blocks of identical size.

SYS_MEDIA_PROPERTY Enumeration

Contains information of property of a media.

File

[system_media.h](#)

C

```
typedef enum {
    SYS_MEDIA_SUPPORTS_BYTE_WRITES = 0x01,
```

```

SYS_MEDIA_SUPPORTS_READ_ONLY = 0x02,
SYS_MEDIA_SUPPORTS_ONE_TIME_PROGRAMMING = 0x04,
SYS_MEDIA_READ_IS_BLOCKING = 0x08,
SYS_MEDIA_WRITE_IS_BLOCKING = 0x10
} SYS_MEDIA_PROPERTY;

```

Members

Members	Description
SYS_MEDIA_SUPPORTS_BYTE_WRITES = 0x01	Media supports Byte Write
SYS_MEDIA_SUPPORTS_READ_ONLY = 0x02	Media supports only Read operation
SYS_MEDIA_SUPPORTS_ONE_TIME_PROGRAMMING = 0x04	Media supports OTP (One Time Programming)
SYS_MEDIA_READ_IS_BLOCKING = 0x08	Read in blocking
SYS_MEDIA_WRITE_IS_BLOCKING = 0x10	Write is blocking

Description

SYS Media Property Structure

This structure contains the information of property of a media device.

Remarks

For a device, if multiple properties are applicable, they can be ORed together and used.

SYS_MEDIA_BLOCK_COMMAND_HANDLE Type

Handle identifying block commands of the media.

File

[system_media.h](#)

C

```
typedef uintptr_t SYS_MEDIA_BLOCK_COMMAND_HANDLE;
```

Description

SYS Media 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.

SYS_MEDIA_BLOCK_EVENT Enumeration

Identifies the possible events that can result from a request.

File

[system_media.h](#)

C

```
typedef enum {
    SYS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE,
```

```

SYS_MEDIA_EVENT_BLOCK_COMMAND_ERROR
} SYS_MEDIA_BLOCK_EVENT;

```

Members

Members	Description
SYS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE	Block operation has been completed successfully.
SYS_MEDIA_EVENT_BLOCK_COMMAND_ERROR	There was an error during the block operation

Description

SYS Media Events

This enumeration identifies the possible events that can result from a media.

Remarks

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

SYS_MEDIA_COMMAND_STATUS Enumeration

The enumeration for status of buffer

File

[system_media.h](#)

C

```

typedef enum {
    SYS_MEDIA_COMMAND_COMPLETED = 0,
    SYS_MEDIA_COMMAND_QUEUED = 1,
    SYS_MEDIA_COMMAND_IN_PROGRESS = 2,
    SYS_MEDIA_COMMAND_UNKNOWN = -1
} SYS_MEDIA_COMMAND_STATUS;

```

Members

Members	Description
SYS_MEDIA_COMMAND_COMPLETED = 0	Done OK and ready
SYS_MEDIA_COMMAND_QUEUED = 1	Scheduled but not started
SYS_MEDIA_COMMAND_IN_PROGRESS = 2	Currently being in transfer
SYS_MEDIA_COMMAND_UNKNOWN = -1	Unknown buffer

Description

Status of buffer

This enumeration contains the various status of buffer.

Remarks

None.

SYS_MEDIA_EVENT_HANDLER Type

Pointer to the Media Event Handler function.

File

[system_media.h](#)

C

```
typedef void (* SYS_MEDIA_EVENT_HANDLER)(SYS_MEDIA_BLOCK_EVENT event,  

SYS_MEDIA_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t context);
```

Returns

None.

Description

Media Event Handler function pointer

This data type defines the required function signature for the media 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.

Remarks

None.

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the media operation requests
context	Value identifying the context of the application that registered the event handling function

SYS_MEDIA_STATUS Enumeration

The state of media.

File

[system_media.h](#)

C

```
typedef enum {
    SYS_MEDIA_DETACHED,
    SYS_MEDIA_ATTACHED
} SYS_MEDIA_STATUS;
```

Members

Members	Description
SYS_MEDIA_DETACHED	Media is detached
SYS_MEDIA_ATTACHED	Media is attached

Description

Status of media

This enumeration states if the media is attached or not

Remarks

None.

Media System

Files

Files

Name	Description
system_common.h	Common System Services definitions and declarations.
system_media.h	System Media declarations and types.
system_module.h	Defines definitions and declarations related to system modules.

Description

This section lists the source and header files used by the library.

system_common.h

Common System Services definitions and declarations.

Enumerations

	Name	Description
	MAIN_RETURN_CODES	Defines return codes for "main".

Macros

	Name	Description
	MAIN_RETURN	Defines the correct return type for the "main" routine.
	MAIN_RETURN_CODE	Casts the given value to the correct type for the return code from "main".
	SYS_ASSERT	Implements default system assert routine, asserts that "test" is true.
	SYS_VersionGet	Gets SYS_COMMON version in numerical format.
	SYS_VersionStrGet	Gets SYS_COMMON version in string format.

Description

System Services Common Library Header

This file provides common system services definitions and declarations.

Remarks

None.

File Name

`sys_common.h`

Company

Microchip Technology Inc.

system_media.h

System Media declarations and types.

Enumerations

	Name	Description
	SYS_MEDIA_BLOCK_EVENT	Identifies the possible events that can result from a request.
	SYS_MEDIA_COMMAND_STATUS	The enumeration for status of buffer
	SYS_MEDIA_PROPERTY	Contains information of property of a media.
	SYS_MEDIA_STATUS	The state of media.

Macros

	Name	Description
	SYS_MEDIA_GEOMETRY_TABLE_ERASE_ENTRY	Erase Region Geometry Table Index Numbers
	SYS_MEDIA_GEOMETRY_TABLE_READ_ENTRY	Read Region Geometry Table Index Numbers
	SYS_MEDIA_GEOMETRY_TABLE_WRITE_ENTRY	Write Region Geometry Table Index Numbers

Structures

	Name	Description
	SYS_MEDIA_GEOMETRY	Contains all the geometrical information of a media device.
	SYS_MEDIA_REGION_GEOMETRY	Contains information of a sys media region.

Types

	Name	Description
	SYS_MEDIA_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the media.
	SYS_MEDIA_EVENT_HANDLER	Pointer to the Media Event Handler function.

Description

System Service Media Interface Declarations and Types

This file contains function and type declarations required to interact with the MPLAB Harmony Media Drivers and File System.

File Name

sys_media.h

Company

Microchip Technology Inc.

system_module.h

Defines definitions and declarations related to system modules.

Enumerations

	Name	Description
	SYS_STATUS	Identifies the current status/state of a system module (including device drivers).

Macros

	Name	Description
	SYS_MODULE_OBJ_INVALID	Object handle value returned if unable to initialize the requested instance of a system module.
	SYS_MODULE_OBJ_STATIC	Object handle value returned by static modules.

Types

	Name	Description
	SYS_MODULE_DEINITIALIZE_ROUTINE	Pointer to a routine that deinitializes a system module (driver, library, or system-maintained application).
	SYS_MODULE_INDEX	Identifies which instance of a system module should be initialized or opened.
	SYS_MODULE_INITIALIZE_ROUTINE	Pointer to a routine that initializes a system module (driver, library, or system-maintained application).
	SYS_MODULE_OBJ	Handle to an instance of a system module.
	SYS_MODULE_REINITIALIZE_ROUTINE	Pointer to a routine that reinitializes a system module (driver, library, or system-maintained application)
	SYS_MODULE_STATUS_ROUTINE	Pointer to a routine that gets the current status of a system module (driver, library, or system-maintained application).

	SYS_MODULE_TASKS_ROUTINE	Pointer to a routine that performs the tasks necessary to maintain a state machine in a module system module (driver, library, or system-maintained application).
--	--	---

Unions

	Name	Description
	SYS_MODULE_INIT	Initializes a module (including device drivers) as requested by the system.

Description

System Module Header

This file defines definitions and interfaces related to system modules.

Remarks

None.

File Name

sys_module.h

Company

Microchip Technology Inc.

Time System Service Library Help

Introduction

This library provides interfaces to manage alarms and/or delays.

Description

The Timer System Service Library is capable of providing periodic or one-shot alarm and delays to the user. It uses a hardware timer peripheral library for providing its timing services. The hardware timer must support compare mode operation.

Key Features:

- Periodic notifications
- One-shot/single notification
- Delays
- Tickless implementation - Instead of getting interrupted from the hardware timer on every configured tick, it manages the hardware timer to generate an interrupt only when required and at appropriate times.

Using the Library

This topic describes the basic architecture of the Time System Service Library and provides information on how it works.

Description

The Time System Service provides alarm and delay functionalities to multiple clients. In addition, it also provides APIs to read the counter value which can be used to measure the time period between two events.

Alarm

- For single/one-shot alarms, the application must register a callback. Once the requested time period elapses, the application is notified by calling the registered callback. After the callback is given, the internal timer object is destroyed by Time System Service, making it available for application to request new alarms/delays.
- For periodic alarms, the application can either register a callback or can choose to poll the status of the alarm. Once the requested time period elapses, the application is notified by calling the callback if registered by the application. In case the application wants to poll the status of the alarm it can do so by calling the status API provided by the library.

- Application can use the delay functionality by requesting a delay using the delay related APIs and then polling the status of the requested delay. Calling the status API after the delay has expired will destroy the internal timer object making it available for new alarms/delays.
- The library also provides counter functionality, that can be used by the application to measure the time elapsed between the two events. The counter functionality does not require a dedicated timer object and is always available.

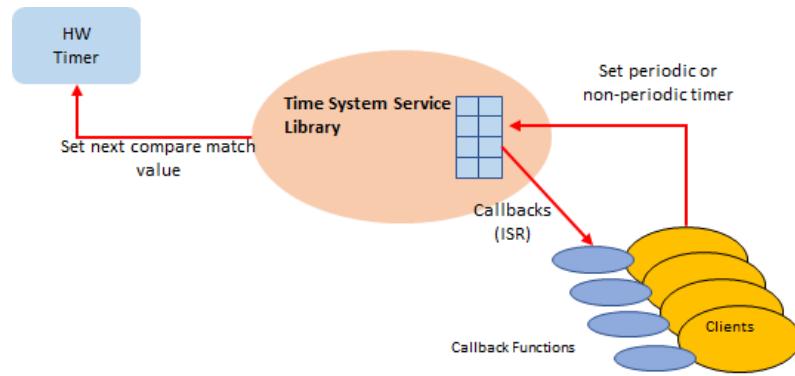
Abstraction Model

The Time System Service library provides an abstraction to the hardware timer to provide following functionalities.

- Periodic Callback
- One Shot/Single Callback
- Delays

Description

The following diagram depicts the Time System Service abstraction model.



How the Library Works

This topic describes the basic architecture of the Time System Service Library and provides information on its implementation.

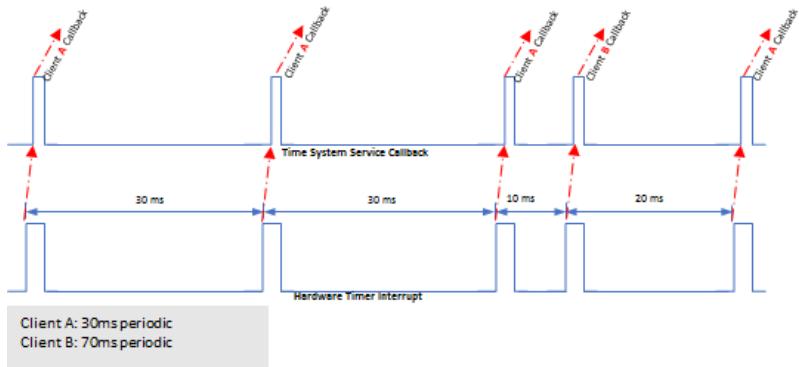
Description

The tickless implementation reduces the overheads of servicing the unneeded hardware timer interrupt on every tick. The tickless implementation can also provide higher resolution compared to a tick based implementation, as there is no fixed tick rate.

Execution Flow

- The Time System Service registers a callback with the underlying hardware timer peripheral library.
- Depending on the alarm/delay time periods requested by various clients, the implementation manages the hardware timer such that a compare interrupt is generated only when needed and at appropriate times.
 - This is achieved by maintaining a sorted list of timing requested by different clients such that the head of the list always indicates the time after which the hardware timer must interrupt and notify the Timer System Service.
- Inside the Time System Service callback, the list is updated and a callback is given to the client for which the alarm/delay has expired.

The following diagram shows how the tickless implementation reduces the number of hardware interrupts. In this example, two clients - Client A and Client B requests a periodic alarm every 30 ms and 70 ms respectively.

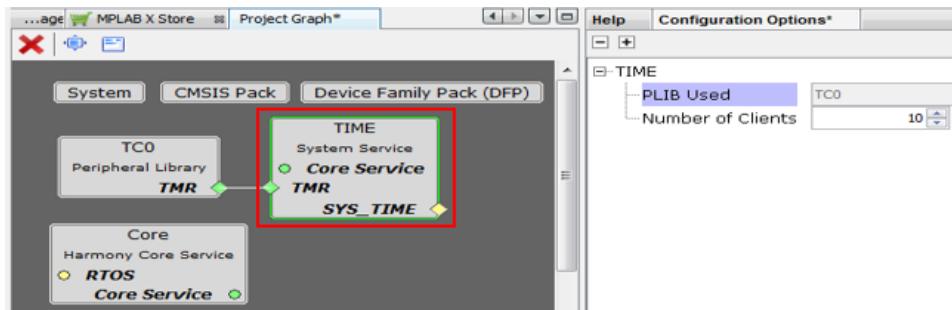


Configuring the Library

This section provides information on how to configure the Time System Service library.

Description

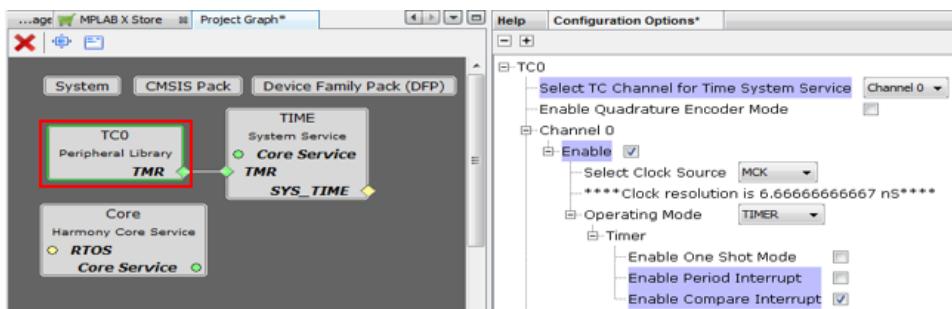
The Time System Service library should be configured via MHC. Below is the snapshot of the MHC configuration window for configuring the Time System Service and a brief description of various configuration options.

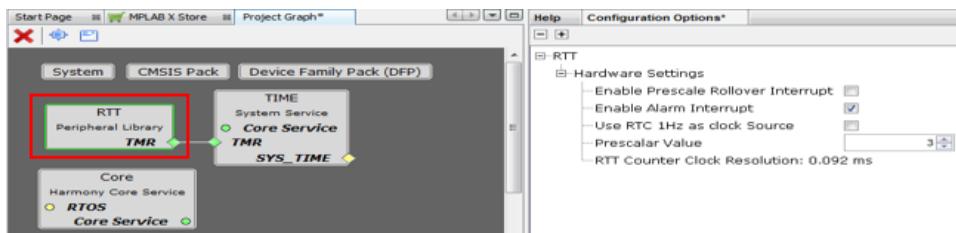


Configuration Options:

- PLIB Used**
 - Indicates the hardware Timer Peripheral Library instance used by the Time System Service.
- Number of Clients**
 - Indicates the maximum number of alarm/delay requests that can be active at any given time.

The hardware Timer Peripheral library is automatically configured for the correct mode of operation when it is connected to the Time System Service as shown in the below snapshots for TC instance 0 and RTT.





Building the Library

This section provides information on how the Time System Service Library can be built.

Description

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Library Interface

a) System Functions

	Name	Description
≡	SYS_TIME_Initialize	Initializes the System Time module.
≡	SYS_TIME_Status	Returns System Time status.
≡	SYS_TIME_Deinitialize	Deinitializes the specific module instance of the SYS TIMER module

b) Timer Functions

	Name	Description
≡	SYS_TIME_TimerCreate	Creates and initializes a new 32-bit software timer instance.
≡	SYS_TIME_TimerDestroy	Destroys/deallocates a software timer instance.
≡	SYS_TIME_TimerStart	Starts a software timer running.
≡	SYS_TIME_TimerStop	Stops a running software timer.
≡	SYS_TIME_TimerCounterGet	Gets the elapsed counter value of a software timer.
≡	SYS_TIME_TimerPeriodHasExpired	Reports whether or not the current period of a software timer has expired.
≡	SYS_TIME_TimerReload	Reloads (or reinitializes) the software timer instance.

c) Callback and Delay Functions

	Name	Description
≡	SYS_TIME_CallbackRegisterMS	Registers a function with the time system service to be called back when the requested number of milliseconds has expired (either once or repeatedly).
≡	SYS_TIME_CallbackRegisterUS	Registers a function with the time system service to be called back when the requested number of microseconds have expired (either once or repeatedly).
≡	SYS_TIME_DelayMS	This function is used to generate a delay of a given number of milliseconds.
≡	SYS_TIME_DelayUS	This function is used to generate a delay of a given number of microseconds.
≡	SYS_TIME_DelayIsComplete	Determines if the given delay timer has completed.

d) Counter and Conversion Functions

	Name	Description
≡	SYS_TIME_Counter64Get	Get the common 64-bit system counter value.
≡	SYS_TIME_CounterGet	Get the common 32-bit system counter value.
≡	SYS_TIME_CounterSet	Sets the common 32-bit system counter value.
≡	SYS_TIME_CountToMS	Converts a counter value to time interval in milliseconds.

≡	SYS_TIME_CountToUS	Converts a counter value to time interval in microseconds.
≡	SYS_TIME_MSToCount	Convert the given time interval in milliseconds to an equivalent counter value.
≡	SYS_TIME_USToCount	Convert the given time interval in microseconds to an equivalent counter value.
≡	SYS_TIME_FrequencyGet	Gets the frequency at which the hardware timer counts.

e) Data Types and Constants

	Name	Description
◆	_SYS_TIME_INIT	TIME system service Initialization Data Declaration
	SYS_TIME_INIT	Defines the data required to initialize the TIME system service
	SYS_TIME_CALLBACK	Pointer to a time system service callback function.
	SYS_TIME_CALLBACK_TYPE	Identifies the type of callback requested (single or periodic).
	SYS_TIME_RESULT	Result of a time service client interface operation.
	SYS_TIME_HANDLE_INVALID	Invalid handle value to a software timer instance.
	SYS_TIME_HANDLE	Handle to a software timer instance.
	SYS_TIME_PLIB_CALLBACK	TIME PLIB API Set needed by the system service
	SYS_TIME_PLIB_CALLBACK_REGISTER	This is type SYS_TIME_PLIB_CALLBACK_REGISTER.
	SYS_TIME_PLIB_COMPARE_SET	This is type SYS_TIME_PLIB_COMPARE_SET.
	SYS_TIME_PLIB_COUNTER_GET	This is type SYS_TIME_PLIB_COUNTER_GET.
	SYS_TIME_PLIB_FREQUENCY_GET	This is type SYS_TIME_PLIB_FREQUENCY_GET.
	SYS_TIME_PLIB_INTERFACE	This is type SYS_TIME_PLIB_INTERFACE.
	SYS_TIME_PLIB_PERIOD_SET	This is type SYS_TIME_PLIB_PERIOD_SET.
	SYS_TIME_PLIB_START	This is type SYS_TIME_PLIB_START.
	SYS_TIME_PLIB_STOP	This is type SYS_TIME_PLIB_STOP.

Description

This section describes the API functions of the Time System Service library.

Refer to each section for a detailed description.

a) System Functions

SYS_TIME_Initialize Function

Initializes the System Time module.

File

[sys_time.h](#)

C

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

Returns

If successful, returns a valid handle to an object. Otherwise, it returns [SYS_MODULE_OBJ_INVALID](#).

Description

This function initializes the instance of the System Time module.

Remarks

This routine should normally only be called once during system initialization.

Example

```

const SYS_TIME_PLIB_INTERFACE sysTimePlibAPI = {
    .timerCallbackSet = (SYS_TIME_PLIB_CALLBACK_REGISTER)TC0_CH0_TimerCallbackRegister,
    .timerCounterGet = (SYS_TIME_PLIB_COUNTER_GET)TC0_CH0_TimerCounterGet,
    .timerPeriodSet = (SYS_TIME_PLIB_PERIOD_SET)TC0_CH0_TimerPeriodSet,
    .timerFrequencyGet = (SYS_TIME_PLIB_FREQUENCY_GET)TC0_CH0_TimerFrequencyGet,
    .timerCompareSet = (SYS_TIME_PLIB_COMPARE_SET)TC0_CH0_TimerCompareSet,
    .timerStart = (SYS_TIME_PLIB_START)TC0_CH0_TimerStart,
    .timerStop = (SYS_TIME_PLIB_STOP)TC0_CH0_TimerStop
};

const SYS_TIME_INIT sysTimeInitData =
{
    .timePlib = &sysTimePlibAPI,
    .hwTimerIntNum = TC0_CH0_IRQn,
};

SYS_MODULE_OBJ objSysTime;

objSysTime = SYS_TIME_Initialize(SYS_TIME_INDEX_0, (SYS_MODULE_INIT *)&sysTimeInitData);
if (objSysTime == SYS_MODULE_OBJ_INVALID)
{
    // Handle error
}

```

Parameters

Parameters	Description
index	Index for the instance to be initialized
init	Pointer to a data structure containing data necessary to initialize the module.

Function

```

SYS_MODULE_OBJ SYS_TIME_Initialize ( const SYS_MODULE_INDEX index,
const           SYS_MODULE_INIT * const init )

```

SYS_TIME_Status Function

Returns System Time status.

File

[sys_time.h](#)

C

```

SYS_STATUS SYS_TIME_Status(SYS_MODULE_OBJ object);

```

Returns

SYS_STATUS_UNINITIALIZED - Indicates that the driver is not initialized.

SYS_STATUS_READY - Indicates that the module initialization is complete and it ready to be used.

Description

This function returns the current status of the System Time module.

Remarks

None.

Preconditions

None.

Example

```
// Handle "objSysTime" value must have been returned from SYS_TIME_Initialize.
if (SYS_TIME_Status (objSysTime) == SYS_STATUS_READY)
{
    // Time system service is initialized and ready to accept new requests.
}
```

Parameters

Parameters	Description
object	SYS TIME object handle, returned from SYS_TIME_Initialize

Function

[SYS_STATUS SYS_TIME_Status \(SYS_MODULE_OBJ object \)](#)

SYS_TIME_Deinitialize Function

Deinitializes the specific module instance of the SYS TIMER module

File

[sys_time.h](#)

C

```
void SYS_TIME_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the specific module instance disabling its operation (and any hardware for driver modules). Resets all of the internal data structures and fields for the specified instance to the default settings.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again.

Preconditions

The [SYS_TIME_Initialize](#) function should have been called before calling this function.

Example

```
// Handle "objSysTime" value must have been returned from SYS_TIME_Initialize.

SYS_TIME_Deinitialize (objSysTime);

if (SYS_TIME_Status (objSysTime) != SYS_STATUS_UNINITIALIZED)
{
    // Check again later if you need to know
    // when the SYS TIME is De-initialized.
}
```

Parameters

Parameters	Description
object	SYS TIMER object handle, returned from SYS_TIME_Initialize

Function

[void SYS_TIME_Deinitialize \(SYS_MODULE_OBJ object \)](#)

b) Timer Functions**SYS_TIME_TimerCreate Function**

Creates and initializes a new 32-bit software timer instance.

File

[sys_time.h](#)

C

```
SYS_TIME_HANDLE SYS_TIME_TimerCreate(uint32_t count, uint32_t period, SYS_TIME_CALLBACK
callback, uintptr_t context, SYS_TIME_CALLBACK_TYPE type);
```

Returns

An opaque value used to identify software timer instance if the call succeeds in allocating/creating the software timer. If the call fails [SYS_TIME_HANDLE_INVALID](#) is returned.

Description

This function creates/allocates a new instance of a 32-bit software timer.

A software timer provides a counter that is separate from other timer counters and is under control of the caller. The counter can be started and stopped under caller control and its counter value and period value can be changed while the counter is either running or stopped.

Remarks

None.

Preconditions

The [SYS_TIME_Initialize](#) function should have been called before calling this function.

Example

Given an implementation of the following function prototype:

```
void MyCallback ( uintptr_t context );
```

The following example creates a software timer instance.

```
SYS_TIME_HANDLE handle;
//myData is the user-defined data that will be passed back in the registered callback function.
handle = SYS_TIME_TimerCreate(0, SYS_TIME_MSToCount(200), &MyCallback, (uintptr_t)&myData,
SYS_TIME_SINGLE);
if (handle != SYS_TIME_HANDLE_INVALID)
{
    //timer is created successfully.
}
```

Parameters

Parameters	Description
count	The initial value of the counter, after the timer has been created and before it has been started.
period	The counter interval at which the timer indicates time has elapsed.
callback	Pointer to function that will be called every time the period counts have elapsed. (Actual timing will depend on system performance and the base frequency at which the time service is configured). For single shot timers, the callback cannot be NULL. For periodic timers, if the callback pointer is given as NULL, no callback will occur, but SYS_TIME_TimerPeriodHasExpired can still be polled to determine if the period has expired for a periodic shot timer.

context	A caller-defined value that's passed (unmodified) back to the client as a parameter of callback function. It can be used to identify the client's context or passed with any value.
type	Type of callback requested. If type is SYS_TIME_SINGLE, the Callback function will be called once when the time period expires. If type is SYS_TIME_PERIODIC Callback function will be called repeatedly, every time the time period expires until the timer object is stopped or deleted.

Function

```
SYS_TIME_HANDLE SYS_TIME_TimerCreate (
    uint32_t count,
    uint32_t period,
    SYS_TIME_CALLBACK callback,
    uintptr_t context,
    SYS_TIME_CALLBACK_TYPE type )
```

SYS_TIME_TimerDestroy Function

Destroys/deallocates a software timer instance.

File

sys_time.h

C

```
SYS_TIME_RESULT SYS_TIME_TimerDestroy(SYS_TIME_HANDLE handle);
```

Returns

SYS_TIME_SUCCESS - If the given software was successfully destroyed.
 SYS_TIME_ERROR - If an error occurred or the given handle was invalid.

Description

This function deletes and deallocates a software timer instance, stopping its counter and releasing the associated resources.

Remarks

Released timer resources can be reused by other clients. Single shot timers are auto destroyed on expiry. Calling [SYS_TIME_DelayIsComplete](#) auto destroys the delay timer if it has expired.

Preconditions

The [SYS_TIME_Initialize](#) and a valid handle to the software timer to be destroyed must be available.

Example

```
// "timer" is the handle to the software timer to be destroyed.
if (SYS_TIME_TimerDestroy(timer) != SYS_TIME_SUCCESS)
{
    // Handle Error
}
```

Parameters

Parameters	Description
handle	Handle to a software timer instance.

Function

```
SYS_TIME_RESULT SYS_TIME_TimerDestroy ( SYS_TIME_HANDLE handle)
```

SYS_TIME_TimerStart Function

Starts a software timer running.

File

[sys_time.h](#)

C

```
SYS_TIME_RESULT SYS_TIME_TimerStart(SYS_TIME_HANDLE handle);
```

Returns

SYS_TIME_SUCCESS if the operation succeeds.

SYS_TIME_ERROR if the operation fails (due, for example, to an invalid handle).

Description

This function starts a previously created software timer.

Remarks

Calling SYS_TIME_TimerStart on an already running timer will have no affect and will return SYS_TIME_SUCCESS. Calling SYS_TIME_TimerStart on a timer that is stopped, will always restart the timer from its initial configured timer/counter value and will not resume the timer from the counter value at which it was stopped.

Preconditions

The [SYS_TIME_Initialize](#) must have been called and a valid handle to the software timer to be started must be available.

Example

Given a "timer" handle, the following example will start the timer's counter running.

```
SYS_TIME_TimerStart(timer);
```

Parameters

Parameters	Description
handle	Handle to a software timer instance.

Function

```
SYS_TIME_RESULT SYS_TIME_TimerStart ( SYS_TIME_HANDLE handle )
```

SYS_TIME_TimerStop Function

Stops a running software timer.

File

[sys_time.h](#)

C

```
SYS_TIME_RESULT SYS_TIME_TimerStop(SYS_TIME_HANDLE handle);
```

Returns

SYS_TIME_SUCCESS if the operation succeeds.

SYS_TIME_ERROR if the operation fails (due, for example, to an invalid handle).

Description

This function stops a previously created and running software timer (i.e. the given timer's counter will stop incrementing).

Remarks

Calling SYS_TIME_TimerStop on a timer that is not running will have no affect and will return SYS_TIME_SUCCESS.

Preconditions

The [SYS_TIME_Initialize](#) must have been called and a valid handle to the software timer to be stopped must be available.

Example

Given a "timer" handle, the following example will stops the timer's counter running.

```
SYS_TIME_TimerStop(timer);
```

Parameters

Parameters	Description
handle	Handle to a software timer instance.

Function

```
SYS_TIME_RESULT SYS_TIME_TimerStop ( SYS_TIME_HANDLE handle )
```

SYS_TIME_TimerCounterGet Function

Gets the elapsed counter value of a software timer.

File

[sys_time.h](#)

C

```
SYS_TIME_RESULT SYS_TIME_TimerCounterGet(SYS_TIME_HANDLE handle, uint32_t * count);
```

Returns

SYS_TIME_SUCCESS if the operation succeeds.

SYS_TIME_ERROR if the operation fails (due, for example, to an to an invalid handle).

Description

This function gets the elapsed counter value of the software timer identified by the handle given.

Remarks

The counter value may be stale immediately upon function return, depending upon timer frequency and system performance.

Preconditions

The [SYS_TIME_Initialize](#) must have been called and a valid handle to the software timer must be available.

Example

Given a "timer" handle, the following example will get the given software timer's elapsed counter value.

```
uint32_t count;
if (SYS_TIME_TimerCounterGet(timer, &count) != SYS_TIME_SUCCESS)
{
    // Handle error
}
```

Parameters

Parameters	Description
handle	Handle to a software timer instance.
count	Address of the variable to receive the value of the given software timer's elapsed counter. This parameter is ignored when the return value is not SYS_TIME_SUCCESS.

Function

```
SYS_TIME_RESULT SYS_TIME_TimerCounterGet (
    SYS_TIME_HANDLE handle,
    uint32_t *count
)
```

SYS_TIME_TimerPeriodHasExpired Function

Reports whether or not the current period of a software timer has expired.

File

[sys_time.h](#)

C

```
bool SYS_TIME_TimerPeriodHasExpired(SYS_TIME_HANDLE handle);
```

Returns

true - If the period has expired.

false - If the period is not expired.

Description

This function reports whether or not the current period of the given software timer has expired and clears the internal flag tracking period expiration so that each period expiration will only be reported once.

Remarks

1. For a periodic timer, a call to `SYS_TIME_TimerPeriodHasExpired` returns true after the first time period has expired. After calling this function, the expiry flag is internally cleared and is set again once the ongoing period of the periodic timer expires.
2. Unlike the `SYS_TIME_DelayIsComplete` routine the `SYS_TIME_TimerPeriodHasExpired` does not delete the timer, it just returns the status of the timer.
3. To poll the status of the delay timers, `SYS_TIME_DelayIsComplete` must be used instead of the `SYS_TIME_TimerPeriodHasExpired` routine, as `SYS_TIME_DelayIsComplete` additionally deletes the delay timer object once the delay has expired.
4. Since single shot timers does not support polling (registration of a callback is mandatory for single shot timers), the `SYS_TIME_CallbackRegisterMS` routine must not be used to poll the status of the single shot timers.

Preconditions

The `SYS_TIME_Initialize` and `SYS_TIME_TimerCreate` or `SYS_TIME_CallbackRegisterMS/SYS_TIME_CallbackRegisterUS` functions (with callback type set to `SYS_TIME_PERIODIC`) must have been called before calling this function.

Example

```
if (SYS_TIME_TimerPeriodHasExpired(timer) == true)
{
    // Timer has expired. Take desired action.
}
```

Parameters

Parameters	Description
handle	Handle to a software timer instance

Function

```
bool SYS_TIME_TimerPeriodHasExpired ( SYS_TIME_HANDLE handle )
```

SYS_TIME_TimerReload Function

Reloads (or reinitializes) the software timer instance.

File

[sys_time.h](#)

C

```
SYS_TIME_RESULT SYS_TIME_TimerReload(SYS_TIME_HANDLE handle, uint32_t count, uint32_t period,
SYS_TIME_CALLBACK callback, uintptr_t context, SYS_TIME_CALLBACK_TYPE type);
```

Returns

SYS_TIME_SUCCESS - If the call succeeded.

SYS_TIME_ERROR - If the call failed (and the timer was not modified).

Description

This function reloads the initial values for an already created/allocated instance of a software timer, even if it is currently running.

Remarks

This function facilitates changing multiple timer parameters quickly and atomically.

Preconditions

The [SYS_TIME_Initialize](#) must have been called and a valid handle to the software timer to be reloaded must be available.

Example

Given an implementation of the following function prototype:

```
void MyNewCallback ( uintptr_t context);
```

The following example updates a software timer instance.

```
//myNewData is the user-defined data that will be passed back in the registered callback
function.
if (SYS_TIME_TimerReload(timer, 0, SYS_TIME_MSToCount(500), &MyNewCallback,
(uintptr_t)&myNewData, SYS_TIME_PERIODIC) != SYS_TIME_SUCCESS )
{
    // Handle error
}
```

Parameters

Parameters	Description
handle	Handle to a software timer instance.
count	The new value of the counter.
period	The new period value.
callback	The new callback function pointer. For single shot timers, the callback must be specified. For periodic timers, if the callback pointer is given as NULL, no callback will occur, but SYS_TIME_TimerPeriodHasExpired can still be polled to determine if the period has expired for a periodic timer.
context	The new caller-defined value that's passed (unmodified) back to the client as a parameter of callback function.
type	Type of callback requested. If type is SYS_TIME_SINGLE, the Callback function will be called once when the time period expires. If type is SYS_TIME_PERIODIC Callback function will be called repeatedly, every time the time period expires until the timer object is stopped or deleted.

Function

```
SYS_TIME_RESULT SYS_TIME_TimerReload (
    SYS_TIME_HANDLE handle,
    uint32_t count,
```

```

    uint32_t period,
    SYS_TIME_CALLBACK callback,
    uintptr_t context,
    SYS_TIME_CALLBACK_TYPE type
)

```

c) Callback and Delay Functions

SYS_TIME_CallbackRegisterMS Function

Registers a function with the time system service to be called back when the requested number of milliseconds has expired (either once or repeatedly).

File

[sys_time.h](#)

C

```
SYS_TIME_HANDLE SYS_TIME_CallbackRegisterMS(SYS_TIME_CALLBACK callback, uintptr_t context,
    uint32_t ms, SYS_TIME_CALLBACK_TYPE type);
```

Returns

[SYS_TIME_HANDLE](#) - A valid timer object handle if the call succeeds. [SYS_TIME_HANDLE_INVALID](#) if it fails.

Description

Creates a timer object and registers a function with it to be called back when the requested delay (specified in milliseconds) has completed. The caller must identify if the timer should call the function once or repeatedly every time the given delay period expires.

Preconditions

The [SYS_TIME_Initialize](#) function should have been called before calling this function.

Example

Given a callback function implementation matching the following prototype:

```
void MyCallback ( uintptr_t context );
```

The following example call will register it, requesting a 50 millisecond periodic callback.

```
//Give a SYS_TIME_CALLBACK function "MyCallback",
SYS_TIME_HANDLE handle = SYS_TIME_CallbackRegisterMS(MyCallback, (uintptr_t)0, 50,
SYS_TIME_PERIODIC);
if (handle != SYS_TIME_HANDLE_INVALID)
{
    //timer is created successfully.
}
```

Parameters

Parameters	Description
callback	Pointer to the function to be called. For single shot timers, the callback cannot be NULL. For periodic timers, if the callback pointer is given as NULL, no callback will occur, but SYS_TIME_TimerPeriodHasExpired can still be polled to determine if the period has expired for a periodic timer.
context	A client-defined value that is passed to the callback function.
ms	Time period in milliseconds.
type	Type of callback requested. If type is SYS_TIME_SINGLE , the Callback function will be called once when the time period expires. If type is SYS_TIME_PERIODIC Callback function will be called repeatedly, every time the time period expires until the timer object is stopped or deleted.

Function

```
SYS_TIME_HANDLE SYS_TIME_CallbackRegisterMS ( SYS_TIME_CALLBACK callback,
    uintptr_t context, uint32_t ms,
    SYS_TIME_CALLBACK_TYPE type )
```

SYS_TIME_CallbackRegisterUS Function

Registers a function with the time system service to be called back when the requested number of microseconds have expired (either once or repeatedly).

File

[sys_time.h](#)

C

```
SYS_TIME_HANDLE SYS_TIME_CallbackRegisterUS(SYS_TIME_CALLBACK callback, uintptr_t context,
    uint32_t us, SYS_TIME_CALLBACK_TYPE type);
```

Returns

SYS_TIME_HANDLE - A valid timer object handle if the call succeeds. **SYS_TIME_HANDLE_INVALID** if it fails.

Description

Creates a timer object and registers a function with it to be called back when the requested delay (specified in microseconds) has completed. The caller must identify if the timer should call the function once or repeatedly every time the given delay period expires.

Preconditions

The **SYS_TIME_Initialize** function should have been called before calling this function.

Example

Given a callback function implementation matching the following prototype:

```
void MyCallback ( uintptr_t context );
```

The following example call will register it, requesting a 500 microsecond periodic callback.

```
//Give a SYS_TIME_CALLBACK function "MyCallback",
SYS_TIME_HANDLE handle = SYS_TIME_CallbackRegisterUS(MyCallback, (uintptr_t)0, 500,
SYS_TIME_PERIODIC);
if (handle != SYS_TIME_HANDLE_INVALID)
{
    //timer is created successfully.
}
```

Parameters

Parameters	Description
callback	Pointer to the function to be called. For single shot timers, the callback cannot be NULL. For periodic timers, if the callback pointer is given as NULL, no callback will occur, but SYS_TIME_TimerPeriodHasExpired can still be polled to determine if the period has expired for a periodic timer.
context	A client-defined value that is passed to the callback function.
us	Time period in microseconds.
type	Type of callback requested. If type is SYS_TIME_SINGLE , the Callback function will be called once when the time period expires. If type is SYS_TIME_PERIODIC Callback function will be called repeatedly, every time the time period expires until the timer object is stopped or deleted.

Function

```
SYS_TIME_HANDLE SYS_TIME_CallbackRegisterUS ( SYS_TIME_CALLBACK callback,
    uintptr_t context, uint32_t us,
    SYS_TIME_CALLBACK_TYPE type )
```

SYS_TIME_DelayMS Function

This function is used to generate a delay of a given number of milliseconds.

File

[sys_time.h](#)

C

```
SYS_TIME_RESULT SYS_TIME_DelayMS(uint32_t ms, SYS_TIME_HANDLE* handle);
```

Returns

SYS_TIME_SUCCESS - If the call succeeded.

SYS_TIME_ERROR - If the call failed.

Description

The function will internally create a single shot timer which will be auto deleted when the application calls [SYS_TIME_DelayIsComplete](#) routine and the delay has expired. The function will return immediately, requiring the caller to use [SYS_TIME_DelayIsComplete](#) routine to check the delay timer's status.

Remarks

Will delay the requested number of milliseconds or longer depending on system performance.

Delay values of 0 will return SYS_TIME_ERROR.

Will return SYS_TIME_ERROR if the timer handle pointer is NULL.

Preconditions

The [SYS_TIME_Initialize](#) function must have been called before calling this function.

Example

```
SYS_TIME_HANDLE timer = SYS_TIME_HANDLE_INVALID;

if (SYS_TIME_DelayMS(100, &timer) != SYS_TIME_SUCCESS)
{
    // Handle error
}
else if (SYS_TIME_DelayIsComplete(timer) != true)
{
    // Wait till the delay has not expired
    while (SYS_TIME_DelayIsComplete(timer) == false);
}
```

Parameters

Parameters	Description
ms	The desired number of milliseconds to delay.
handle	Address of the variable to receive the timer handle value.

Function

[SYS_TIME_RESULT SYS_TIME_DelayMS \(uint32_t ms, SYS_TIME_HANDLE* handle \)](#)

SYS_TIME_DelayUS Function

This function is used to generate a delay of a given number of microseconds.

File

[sys_time.h](#)

C

```
SYS_TIME_RESULT SYS_TIME_DelayUS(uint32_t us, SYS_TIME_HANDLE* handle);
```

Returns

SYS_TIME_SUCCESS - If the call succeeded.

SYS_TIME_ERROR - If the call failed.

Description

The function will internally create a single shot timer which will be auto deleted when the application calls [SYS_TIME_DelayIsComplete](#) routine and the delay has expired. The function will return immediately, requiring the caller to use [SYS_TIME_DelayIsComplete](#) routine to check the delay timer's status.

Remarks

Will delay the requested number of microseconds or longer depending on system performance.

Delay values of 0 will return SYS_TIME_ERROR.

Will return SYS_TIME_ERROR if timer handle pointer is NULL.

Preconditions

The [SYS_TIME_Initialize](#) function must have been called before calling this function.

Example

```
SYS_TIME_HANDLE timer = SYS_TIME_HANDLE_INVALID;

if (SYS_TIME_DelayUS(50, &timer) != SYS_TIME_SUCCESS)
{
    // Handle error
}
else if (SYS_TIME_DelayIsComplete(timer) != true)
{
    // Wait till the delay has not expired
    while (SYS_TIME_DelayIsComplete(timer) == false);
}
```

Parameters

Parameters	Description
us	The desired number of microseconds to delay.
handle	Address of the variable to receive the timer handle value.

Function

```
SYS_TIME_RESULT SYS_TIME_DelayUS ( uint32_t us, SYS_TIME_HANDLE* handle )
```

SYS_TIME_DelayIsComplete Function

Determines if the given delay timer has completed.

File

[sys_time.h](#)

C

```
bool SYS_TIME_DelayIsComplete(SYS_TIME_HANDLE handle);
```

Returns

true - If the delay has completed.

false - If the delay has not completed.

Description

This function determines if the requested delay is completed or is still in progress.

Remarks

SYS_TIME_DelayIsComplete must be called to poll the status of the delay requested through [SYS_TIME_DelayMS](#) or [SYS_TIME_DelayUS](#).

SYS_TIME_DelayIsComplete must not be used to poll the status of a periodic timer. Status of a periodic timer may be polled using the [SYS_TIME_TimerPeriodHasExpired](#) routine.

Preconditions

A delay request must have been created using either the [SYS_TIME_DelayMS](#) or [SYS_TIME_DelayUS](#) functions.

Example

```
// Check if the delay has expired.
if (SYS_TIME_DelayIsComplete(timer) != true)
{
    // Delay has not expired
}
```

Parameters

Parameters	Description
handle	A SYS_TIME_HANDLE value provided by either SYS_TIME_DelayMS or SYS_TIME_DelayUS functions.

Function

bool SYS_TIME_DelayIsComplete ([SYS_TIME_HANDLE](#) handle)

d) Counter and Conversion Functions

SYS_TIME_Counter64Get Function

Get the common 64-bit system counter value.

File

[sys_time.h](#)

C

```
uint64_t SYS_TIME_Counter64Get();
```

Returns

The current "live" value of the common 64-bit system counter.

Description

Returns the current "live" value of the common 64-bit system counter.

Remarks

The value returned may be stale as soon as it is provided, as the timer is live and running at full frequency resolution (as configured and as reported by the [SYS_TIME_FrequencyGet](#) function). If additional accuracy is required, use a hardware timer instance.

Preconditions

The [SYS_TIME_Initialize](#) function should have been called before calling this function.

Example

```
uint64_t timeStamp1;
uint64_t timeStamp2;
uint32_t timeDiffMs;

// Take first time stamp
timeStamp1 = SYS_TIME_Counter64Get();

//Perform some tasks.....

// Take second time stamp
timeStamp2 = SYS_TIME_Counter64Get();

//Calculate the time difference. Use the appropriate function -
//SYS_TIME_CountToMS or SYS_TIME_CountToUS to calculate the time difference.

timeDiffMs = SYS_TIME_CountToMS((timeStamp2 - timeStamp1));
```

Function

```
uint64_t SYS_TIME_Counter64Get ( void )
```

SYS_TIME_CounterGet Function

Get the common 32-bit system counter value.

File

[sys_time.h](#)

C

```
uint32_t SYS_TIME_CounterGet();
```

Returns

The current "live" value of the common 32-bit system counter.

Description

Returns the current "live" value of the common 32-bit system counter.

Remarks

The value returned may be stale as soon as it is provided, as the timer is live and running at full frequency resolution (as configured and as reported by the [SYS_TIME_FrequencyGet](#) function). If additional accuracy is required, use a hardware timer instance.

Preconditions

The [SYS_TIME_Initialize](#) function should have been called before calling this function.

Example

```
uint32_t timeStamp1;
uint32_t timeStamp2;
uint32_t timeDiffMs;

// Take first time stamp
timeStamp1 = SYS_TIME_CounterGet();

//Perform some tasks.....

// Take second time stamp
timeStamp2 = SYS_TIME_CounterGet();

//Calculate the time difference. Use the appropriate function -
//SYS_TIME_CountToMS or SYS_TIME_CountToUS to calculate the time difference.
```

```
timeDiffMs = SYS_TIME_CountToMS((timeStamp2 - timeStamp1));
```

Function

```
uint32_t SYS_TIME_CounterGet ( void )
```

SYS_TIME_CounterSet Function

Sets the common 32-bit system counter value.

File

[sys_time.h](#)

C

```
void SYS_TIME_CounterSet(uint32_t count);
```

Returns

None

Description

Sets the current "live" value of the common 32-bit system counter.

Remarks

None.

Preconditions

The [SYS_TIME_Initialize](#) function should have been called before calling this function.

Example

```
SYS_TIME_CounterSet(1000);
```

Parameters

Parameters	Description
count	The 32-bit counter value to write to the common system counter.

Function

```
void SYS_TIME_CounterSet ( uint32_t count )
```

SYS_TIME_CountToMS Function

Converts a counter value to time interval in milliseconds.

File

[sys_time.h](#)

C

```
uint32_t SYS_TIME_CountToMS(uint32_t count);
```

Returns

Number of milliseconds represented by the given counter value.

Description

This function converts a counter value to time interval in milliseconds, based on the hardware timer frequency as configured and as reported by [SYS_TIME_FrequencyGet](#).

Remarks

None.

Preconditions

The [SYS_TIME_Initialize](#) function should have been called before calling this function.

Example

Given a previously captured `uint32_t` counter value called "timestamp" captured using the [SYS_TIME_CounterGet](#) function, the following example will calculate number of milliseconds elapsed since timestamp was captured.

```
uint32_t count = SYS_TIME_CounterGet() - timestamp;
uint32_t ms    = SYS_TIME_CountToMS(count);
```

Parameters

Parameters	Description
count	Counter value to be converted to an equivalent value in milliseconds.

Function

```
uint32_t SYS_TIME_CountToMS ( uint32_t count )
```

SYS_TIME_CountToUS Function

Converts a counter value to time interval in microseconds.

File

[sys_time.h](#)

C

```
uint32_t SYS_TIME_CountToUS(uint32_t count);
```

Returns

Number of microseconds represented by the given counter value.

Description

This function converts a counter value to time interval in microseconds, based on the hardware timer frequency as configured and as reported by [SYS_TIME_FrequencyGet](#).

Remarks

None.

Preconditions

The [SYS_TIME_Initialize](#) function should have been called before calling this function.

Example

Given a previously captured 32-bit counter value called "timestamp" captured using the [SYS_TIME_CounterGet](#) function, the following example will calculate the number of microseconds elapsed since timestamp was captured.

```
uint32_t count = SYS_TIME_CounterGet() - timestamp;
uint32_t us    = SYS_TIME_CountToUS(count);
```

Parameters

Parameters	Description
count	Counter value to be converted to an equivalent value in microseconds.

Function

```
uint32_t SYS_TIME_CountToUS ( uint32_t count )
```

SYS_TIME_MSToCount Function

Convert the given time interval in milliseconds to an equivalent counter value.

File

[sys_time.h](#)

C

```
uint32_t SYS_TIME_MSToCount(uint32_t ms);
```

Returns

Number of hardware timer counts that will expire in the given time interval.

Description

This function converts a given time interval (measured in milliseconds) to an equivalent 32-bit counter value, based on the configured hardware timer frequency as reported by [SYS_TIME_FrequencyGet](#).

Remarks

None.

Preconditions

The [SYS_TIME_Initialize](#) function should have been called before calling this function.

Example

```
uint32_t futureCounter = SYS_TIME_CounterGet() + SYS_TIME_MSToCount(10);
```

Parameters

Parameters	Description
ms	Time interval in milliseconds.

Function

```
uint32_t SYS_TIME_MSToCount ( uint32_t ms )
```

SYS_TIME_USToCount Function

Convert the given time interval in microseconds to an equivalent counter value.

File

[sys_time.h](#)

C

```
uint32_t SYS_TIME_USToCount(uint32_t us);
```

Returns

Number of hardware timer counts that will expire in the given time interval.

Description

This function converts a given time interval (measured in microseconds) to an equivalent 32-bit counter value, based on the configured hardware timer frequency as reported by [SYS_TIME_FrequencyGet](#).

Remarks

None.

Preconditions

The [SYS_TIME_Initialize](#) function should have been called before calling this function.

Example

```
uint32_t futureCounter = SYS_TIME_CounterGet() + SYS_TIME_USToCount(200);
```

Parameters

Parameters	Description
us	Time interval in microseconds.

Function

```
uint32_t SYS_TIME_USToCount ( uint32_t us )
```

SYS_TIME_FrequencyGet Function

Gets the frequency at which the hardware timer counts.

File

[sys_time.h](#)

C

```
uint32_t SYS_TIME_FrequencyGet();
```

Returns

The frequency at which the hardware timer runs, if the timer has been initialized and is ready. Otherwise, it returns 0.

Description

Returns the frequency at which the hardware timer runs. This frequency determines the maximum resolution of all services provided by SYS_TIME.

Remarks

This frequency is determined by hardware capabilities and how they are configured and initialized.

Preconditions

The [SYS_TIME_Initialize](#) function should have been called before calling this function.

Example

```
uint32_t frequency = SYS_TIME_FrequencyGet();
```

Function

```
uint32_t SYS_TIME_FrequencyGet ( void )
```

e) Data Types and Constants**_SYS_TIME_INIT Structure****File**

[sys_time_definitions.h](#)

C

```
struct _SYS_TIME_INIT {
    const SYS_TIME_PLIB_INTERFACE* timePlib;
    INT_SOURCE hwTimerIntNum;
};
```

Members

Members	Description
const SYS_TIME_PLIB_INTERFACE* timePlib;	Identifies the PLIB API set to be used by the system service to access <ul style="list-style-type: none"> the peripheral.
INT_SOURCE hwTimerIntNum;	Interrupt source ID for the TIMER interrupt.

Description

TIME system service Initialization Data Declaration

SYS_TIME_INIT Type

Defines the data required to initialize the TIME system service

File

[sys_time.h](#)

C

```
typedef struct _SYS_TIME_INIT SYS_TIME_INIT;
```

Description

TIME System Service Initialization Data

This data type defines the data required to initialize the TIME system service.

Remarks

This structure is implementation specific. It is fully defined in [sys_time_definitions.h](#).

SYS_TIME_CALLBACK Type

Pointer to a time system service callback function.

File

[sys_time.h](#)

C

```
typedef void (* SYS_TIME_CALLBACK)(uintptr_t context);
```

Returns

None.

Description

This data type defines a pointer to a time service callback function, thus defining the function signature. Callback functions may be registered by clients of the time service either when creating a software timer or using the `SYS_TIME_Callback` shortcut functions.

Remarks

None.

Preconditions

The time service must have been initialized using the `SYS_TIME_Initialize` function before attempting to register a SYS Time callback function.

Example

```
/**"MY_APP_DATA_STRUCT" is a user defined data structure with "isTimerExpired" as
//one of the structure members.
void MyCallback ( uintptr_t context )
{
```

```

MY_APP_DATA_STRUCT* pAppData = (MY_APP_DATA_STRUCT *) context;
if (pAppData != NULL)
{
    pAppData->isTimerExpired = true;
}

```

Parameters

Parameters	Description
context	A context value, returned untouched to the client when the callback occurs. It can be used to identify the instance of the client who registered the callback.

Function

```
void (* SYS_TIME_CALLBACK ) ( uintptr_t context )
```

SYS_TIME_CALLBACK_TYPE Enumeration

Identifies the type of callback requested (single or periodic).

File

[sys_time.h](#)

C

```

typedef enum {
    SYS_TIME_SINGLE,
    SYS_TIME_PERIODIC
} SYS_TIME_CALLBACK_TYPE;

```

Members

Members	Description
SYS_TIME_SINGLE	Requesting a single (one time) callback.
SYS_TIME_PERIODIC	Requesting a periodically repeating callback.

Description

System Time Callback Type

Remarks

None.

SYS_TIME_RESULT Enumeration

Result of a time service client interface operation.

File

[sys_time.h](#)

C

```

typedef enum {
    SYS_TIME_SUCCESS,
    SYS_TIME_ERROR
} SYS_TIME_RESULT;

```

Members

Members	Description
SYS_TIME_SUCCESS	Operation completed with success.
SYS_TIME_ERROR	Invalid handle or operation failed.

Description

System Time Result

Identifies the result of certain time service operations.

SYS_TIME_HANDLE_INVALID Macro

Invalid handle value to a software timer instance.

File

[sys_time.h](#)

C

```
#define SYS_TIME_HANDLE_INVALID ((SYS_TIME_HANDLE) (-1))
```

Description

Invalid System Time handle value to a software timer

Defines the invalid handle value to a timer instance.

Remarks

Do not rely on the actual value as it may change in different versions or implementations of the SYS Time service.

SYS_TIME_HANDLE Type

Handle to a software timer instance.

File

[sys_time.h](#)

C

```
typedef uintptr_t SYS_TIME_HANDLE;
```

Description

System Time Handle

This data type is a handle to a software timer instance. It can be used to access and control a software timer.

Remarks

Do not rely on the underlying type as it may change in different versions or implementations of the SYS Time service.

SYS_TIME_PLIB_CALLBACK Type

File

[sys_time_definitions.h](#)

C

```
typedef void (* SYS_TIME_PLIB_CALLBACK)(uint32_t , uintptr_t);
```

Description

TIME PLIB API Set needed by the system service

SYS_TIME_PLIB_CALLBACK_REGISTER Type

File

[sys_time_definitions.h](#)

C

```
typedef void (* SYS_TIME_PLIB_CALLBACK_REGISTER)(SYS_TIME_PLIB_CALLBACK callback, uintptr_t context);
```

Description

This is type SYS_TIME_PLIB_CALLBACK_REGISTER.

SYS_TIME_PLIB_COMPARE_SET Type

File

[sys_time_definitions.h](#)

C

```
typedef void (* SYS_TIME_PLIB_COMPARE_SET)(uint32_t compare);
```

Description

This is type SYS_TIME_PLIB_COMPARE_SET.

SYS_TIME_PLIB_COUNTER_GET Type

File

[sys_time_definitions.h](#)

C

```
typedef uint32_t (* SYS_TIME_PLIB_COUNTER_GET)(void);
```

Description

This is type SYS_TIME_PLIB_COUNTER_GET.

SYS_TIME_PLIB_FREQUENCY_GET Type

File

[sys_time_definitions.h](#)

C

```
typedef uint32_t (* SYS_TIME_PLIB_FREQUENCY_GET)(void);
```

Description

This is type SYS_TIME_PLIB_FREQUENCY_GET.

SYS_TIME_PLIB_INTERFACE Structure

File

[sys_time_definitions.h](#)

C

```
typedef struct {
    SYS_TIME_PLIB_CALLBACK_REGISTER timerCallbackSet;
    SYS_TIME_PLIB_PERIOD_SET timerPeriodSet;
    SYS_TIME_PLIB_FREQUENCY_GET timerFrequencyGet;
    SYS_TIME_PLIB_COMPARE_SET timerCompareSet;
    SYS_TIME_PLIB_START timerStart;
    SYS_TIME_PLIB_STOP timerStop;
    SYS_TIME_PLIB_COUNTER_GET timerCounterGet;
} SYS_TIME_PLIB_INTERFACE;
```

Description

This is type SYS_TIME_PLIB_INTERFACE.

SYS_TIME_PLIB_PERIOD_SET Type**File**

[sys_time_definitions.h](#)

C

```
typedef void (* SYS_TIME_PLIB_PERIOD_SET)(uint32_t period);
```

Description

This is type SYS_TIME_PLIB_PERIOD_SET.

SYS_TIME_PLIB_START Type**File**

[sys_time_definitions.h](#)

C

```
typedef void (* SYS_TIME_PLIB_START)(void);
```

Description

This is type SYS_TIME_PLIB_START.

SYS_TIME_PLIB_STOP Type**File**

[sys_time_definitions.h](#)

C

```
typedef void (* SYS_TIME_PLIB_STOP)(void);
```

Description

This is type SYS_TIME_PLIB_STOP.

Files

Files

Name	Description
sys_time.h	Time system service library interface.
sys_time_definitions.h	TIME System Service Definitions Header File

Description

This section will list only the library's interface header file(s).

sys_time.h

Time system service library interface.

Enumerations

	Name	Description
	SYS_TIME_CALLBACK_TYPE	Identifies the type of callback requested (single or periodic).
	SYS_TIME_RESULT	Result of a time service client interface operation.

Functions

	Name	Description
≡◊	SYS_TIME_CallbackRegisterMS	Registers a function with the time system service to be called back when the requested number of milliseconds has expired (either once or repeatedly).
≡◊	SYS_TIME_CallbackRegisterUS	Registers a function with the time system service to be called back when the requested number of microseconds have expired (either once or repeatedly).
≡◊	SYS_TIME_Counter64Get	Get the common 64-bit system counter value.
≡◊	SYS_TIME_CounterGet	Get the common 32-bit system counter value.
≡◊	SYS_TIME_CounterSet	Sets the common 32-bit system counter value.
≡◊	SYS_TIME_CountToMS	Converts a counter value to time interval in milliseconds.
≡◊	SYS_TIME_CountToUS	Converts a counter value to time interval in microseconds.
≡◊	SYS_TIME_Deinitialize	Deinitializes the specific module instance of the SYS TIMER module
≡◊	SYS_TIME_DelayIsComplete	Determines if the given delay timer has completed.
≡◊	SYS_TIME_DelayMS	This function is used to generate a delay of a given number of milliseconds.
≡◊	SYS_TIME_DelayUS	This function is used to generate a delay of a given number of microseconds.
≡◊	SYS_TIME_FrequencyGet	Gets the frequency at which the hardware timer counts.
≡◊	SYS_TIME_Initialize	Initializes the System Time module.
≡◊	SYS_TIME_MSToCount	Convert the given time interval in milliseconds to an equivalent counter value.
≡◊	SYS_TIME_Status	Returns System Time status.
≡◊	SYS_TIME_TimerCounterGet	Gets the elapsed counter value of a software timer.
≡◊	SYS_TIME_TimerCreate	Creates and initializes a new 32-bit software timer instance.
≡◊	SYS_TIME_TimerDestroy	Destroys/deallocates a software timer instance.
≡◊	SYS_TIME_TimerPeriodHasExpired	Reports whether or not the current period of a software timer has expired.
≡◊	SYS_TIME_TimerReload	Reloads (or reinitializes) the software timer instance.
≡◊	SYS_TIME_TimerStart	Starts a software timer running.
≡◊	SYS_TIME_TimerStop	Stops a running software timer.
≡◊	SYS_TIME_USToCount	Convert the given time interval in microseconds to an equivalent counter value.

Macros

	Name	Description
	SYS_TIME_HANDLE_INVALID	Invalid handle value to a software timer instance.

Types

	Name	Description
	SYS_TIME_CALLBACK	Pointer to a time system service callback function.
	SYS_TIME_HANDLE	Handle to a software timer instance.
	SYS_TIME_INIT	Defines the data required to initialize the TIME system service

Description

Time System Service Library Interface Header File

This file defines the interface to the Time system service library. This library provides a free-running shared software timer/counter, (32-bit) giving the entire system a common time base and providing real-time capabilities such as delays and callbacks. It also (optionally) provides the ability to create individual software timers (32-bit), under control of the client, with counter and callback capabilities.

Remarks

This interface will be extended in the future to utilize Real Time Clock and Calendar (RTCC) support to provide time of day and date capabilities.

File Name

sys_time.h

Company

Microchip Technology Inc.

sys_time_definitions.h

TIME System Service Definitions Header File

Structures

	Name	Description
	_SYS_TIME_INIT	TIME system service Initialization Data Declaration
	SYS_TIME_PLIB_INTERFACE	This is type SYS_TIME_PLIB_INTERFACE.

Types

	Name	Description
	SYS_TIME_PLIB_CALLBACK	TIME PLIB API Set needed by the system service
	SYS_TIME_PLIB_CALLBACK_REGISTER	This is type SYS_TIME_PLIB_CALLBACK_REGISTER.
	SYS_TIME_PLIB_COMPARE_SET	This is type SYS_TIME_PLIB_COMPARE_SET.
	SYS_TIME_PLIB_COUNTER_GET	This is type SYS_TIME_PLIB_COUNTER_GET.
	SYS_TIME_PLIB_FREQUENCY_GET	This is type SYS_TIME_PLIB_FREQUENCY_GET.
	SYS_TIME_PLIB_PERIOD_SET	This is type SYS_TIME_PLIB_PERIOD_SET.
	SYS_TIME_PLIB_START	This is type SYS_TIME_PLIB_START.
	SYS_TIME_PLIB_STOP	This is type SYS_TIME_PLIB_STOP.

Description

TIME System Service Definitions Header File

This file provides implementation-specific definitions for the TIME system service's system interface.

File Name

sys_time_definitions.h

Company

Microchip Technology Inc.

OSAL Library Help

This section describes the Operating System Abstraction Layer (OSAL) that is available in MPLAB Harmony.

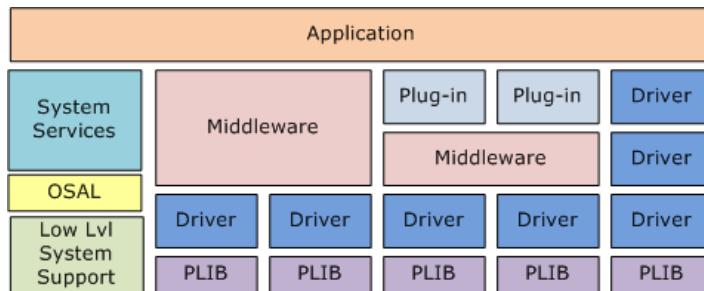
Introduction

The Operating System Abstraction Layer (OSAL) provides a consistent interface to allow MPLAB Harmony-compliant libraries to take advantage of Operating System constructs when running in an OS environment or when operating without one. It is designed to take care of the underlying differences between the available OS Kernels or when no kernel is present.

Description

The OSAL provides the interface to commonly available Real-Time Operating Systems (RTOS) such that MPLAB Harmony libraries may be written using a single interface to a minimal set of OS features needed to provide thread safety. The OSAL interface can be implemented appropriately to support almost any desired RTOS. For systems where no RTOS is available, or desired, a bare version of the OSAL supports either polled or interrupt-driven environments running directly on the hardware. This allows applications designed using MPLAB Harmony libraries to be executed in all three common embedded environments: polled (shared multi-tasking), interrupt-driven, or RTOS-based.

 **Note:** It is possible to make RTOS independent applications using the OSAL. However, as explained in the following section, that is not its purpose. Use and selection of an RTOS is usually determined by the availability of its unique features. And, utilizing those features will, of course, make an application OS-specific.



Scope

By design, the OSAL is a minimal API intended only to enable thread-safe operation for MPLAB Harmony libraries. It only exposes a very small subset of the capabilities of an operating system so that MPLAB Harmony libraries can use semaphores, mutexes, and critical sections (and a few other things) necessary to protect shared resources (data structures, peripheral registers, and other memory objects) from corruption by unsynchronized access by multiple threads. This is done to allow MPLAB Harmony libraries to be made compatible with the largest variety of operating systems, by using a minimal subset of some of the most common OS features. The OSAL is not intended to provide a complete abstraction of an RTOS, which is what you would normally do to implement a complete application. Abstracting an entire operating system is a much more complex task that is roughly equivalent to defining your own RTOS.

The OSAL is not designed to replace a commercial kernel, and therefore, the user is encouraged to use any of the specific features of their chosen RTOS in order to achieve best performance. As such, the OSAL can be considered to be an Operating System Compatibility Layer offering MPLAB Harmony-compliant libraries the required common functions to ensure correct operation in both RTOS and non-RTOS environments.

The common interface presented by the OSAL is designed to offer a set of services typically found on micro-kernel and mini-scheduler systems. Because of this it has no aspirations to provide an equivalent set of capabilities as those found on large multi-tasking systems such as μ CLinux™. The common services are designed to allow MPLAB Harmony to implement thread-safe Drivers and Middleware. The design intention is that drivers will use the minimal set of OSAL features necessary to ensure that they can safely operate in a multi-threaded environment yet can also compile and run correctly when no underlying RTOS is present. The range of features used by a driver is typically limited to these OSAL features (see the [Library Interface](#) section):

- Semaphore Functions
- Mutex Functions
- Critical Section Functions

Supported RTOS

RTOS	Release Type
FreeRTOS v10.0.1	Production

Using the Library

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

Description

Interface Header File: [osal.h](#)

The interface to the OSAL Library is defined in the [osal.h](#) header file. Any C language source (.c) file that uses the OSAL System Service library should include [osal.h](#).

Library File: `osal_<vendor-specified RTOS name>.c` (i.e., FreeRTOS, etc.)

The OSAL Library consists of a basic implementation and individual ports of the OSAL to target operating systems. The basic implementation is used when any of the Third-Party Library RTOS is not instantiated.

When an RTOS is being used (i.e. if any of Third-Party Library RTOS is instantiated, Ex: FreeRTOS) then an external implementation file which provides the required interface wrappers should be added to the project. For instance for the FreeRTOS operating system the file `osal_freertos.c` should be added, while for the Micrium µC/OS-III operating system the file `osal_uicos3.c` should be added.

The basic implementation and some generic ports are provided with the Library, however, it is the responsibility of third-party vendors to supply an implementation file for operating systems that are not already supported.

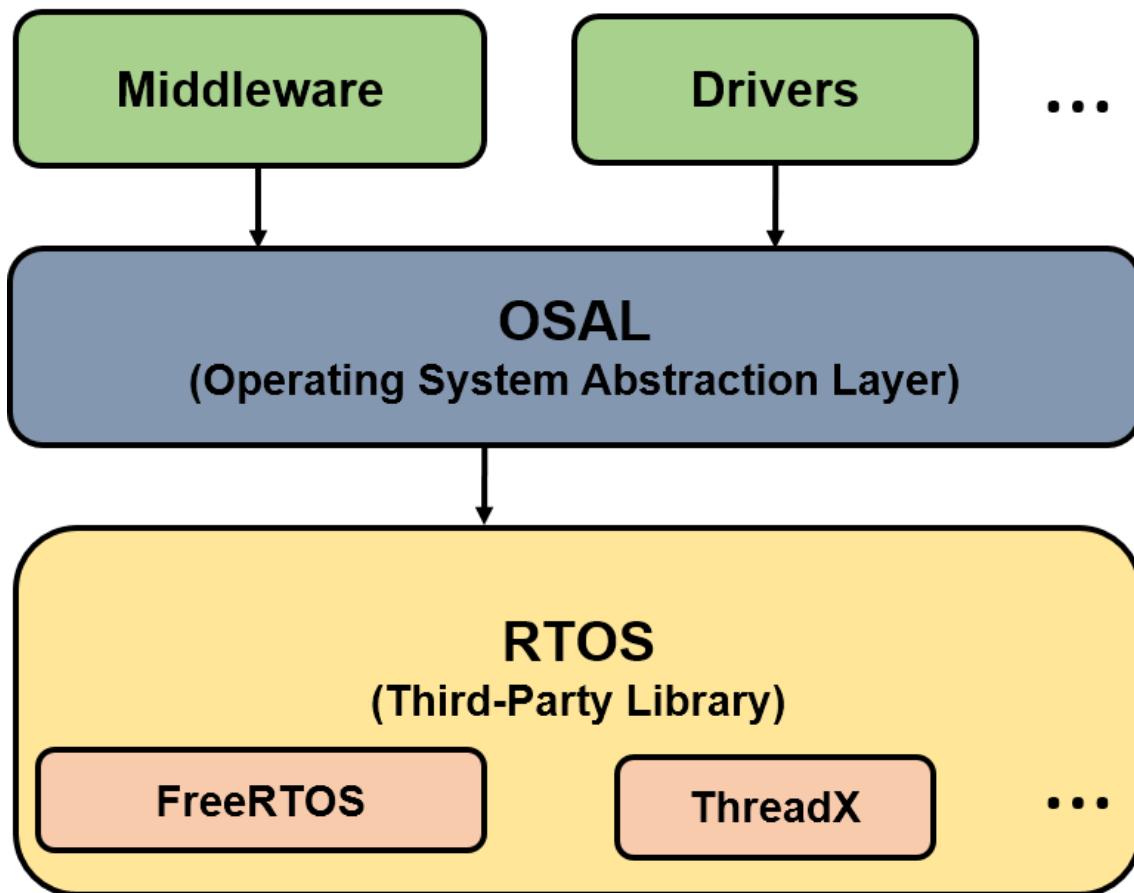
Abstraction Model

The OSAL Library provides a predefined set of functions and types that match common synchronization and communication services that an RTOS will typically provide. It is designed to be a lightweight abstraction model and deliberately excludes the much broader depth and breadth of services that a fully fledged RTOS provides. As such the interface defines only those core functions necessary for the MPLAB Harmony Drivers and middleware to operate in a multi-threaded environment.

Description

The common interface can easily be ported to many host Real-Time Operating Systems (RTOS) by third parties and the set of functions provides a basic level of RTOS compatibility. Where a specific RTOS does not implement a given architectural feature (e.g., events), the OSAL port for that RTOS should endeavor to imitate that feature using the constructs that are available. Although it is recognized that this may have a detrimental effect on the performance of that system it does allow MPLAB Harmony developers the broadest scope for using RTOS features in their designs.

The following diagrams illustrate the OSAL Abstraction model.



Library Overview

This section provides an overview of the OSAL Library.

Description

The OSAL Library provides a defined interface such that driver and middleware developers will be able to create MPLAB Harmony code that can safely operate in a multi-threaded environment when a supported RTOS is present yet will still compile and function correctly when MPLAB Harmony is being used in a non-RTOS environment with an interrupt or non-interrupt driven application model.

At the application layer, the developer is encouraged to use the specific features of a chosen RTOS once it has been selected since this is likely to provide a more effective and rich programming environment.

The OSAL Library is deliberately designed to be a thin layer over an underlying RTOS, which presents a predefined interface to the common features used by the majority of Real-Time Operating Systems, which includes:

Library Interface Section	Description
Semaphore Functions	Binary and counting semaphores.
Mutex Functions	Thread and resource locking mechanism.
Critical Section Functions	Application and scheduler locking mechanism.
Memory Allocation Functions	Memory allocation primitives or wrappers.
OSAL Control Functions	OSAL initialization.

One of the primary design guidelines is that a host operating system may not be present and so any operations that the OSAL presents are designed to compile out to safe default implementations if no RTOS is present. This can mean the following:

- Implementing a dummy function that mimics typical RTOS behavior
- Implementing a `#define` or inline function that returns a 'safe' generic return value, such as 'true' or a 'call succeeded' status

- Returning a OSAL_RESULT_NOT_IMPLEMENTED value to indicate an unsupported operation
- Throwing an OSAL_ASSERT failure to indicate a terminal error that prevents operation under specific circumstances

How the Library Works

This section provides information on how the OSAL Library works.

Semaphores

The semaphore implements a method for thread synchronization. This synchronization can be either between one thread and another or between an ISR and a thread. A semaphore once signalled will unlock the highest priority thread currently pending on it.

Description

A semaphore can be used to lock a shared resource, although it is more normal to use a mutex for such an activity. Once obtained a semaphore should be posted back to enable it to be retaken at a later time or in another thread.

```
/* mainline code prior to OS start */
    /* declare a variable of type semaphore handle */
    OSAL_SEM_DECLARE(semSync);
    /* create the semaphore */
    OSAL_SEM_Create(&semSync, OSAL_SEM_TYPE_BINARY, 0, 0);

/* thread one */
    ...
    /* take the semaphore without waiting */
    OSAL_SEM_Pend(semSync, 0);
    ... perform some actions
    /* return the semaphore */
    OSAL_SEM_Post(semSync);
    ...

/* thread two must not execute until thread one has finished its operations*/
    ...
    /* block on the semaphore */
    OSAL_SEM_Pend(semSync, OSAL_WAIT_FOREVER);
    ... perform some more actions
    /* return the semaphore */
    OSAL_SEM_Post(semSync);
```

A semaphore can be signalled multiple times and so provides a method for an ISR to release a thread waiting on it. Even though the blocked thread never returns the semaphore, because the asynchronous ISR repeatedly posts it the next time the thread wants to pend on the semaphore it will be available. By moving the majority of interrupt service processing from the ISR to a high priority thread the system response time is improved and the eventual processing can take advantage of OSAL features such as mutexes and queues which would normally be harder to implement inside the ISR. This technique is known as deferred interrupt processing.

```
/* an example interrupt handler called from an ISR that performs task synchronization using a
semaphore */
void _ISRTasksRX(void) /* N.B. pseudo-code ISR */
{
    ...

    _DRV_USART_InterruptSourceStatusClear(_DRV_USART_GET_INT_SRC_RX(_DRV_USART_OBJ(dObj,
rxInterruptSource)));

    /* Release the receive semaphore unblocking any tasks */
    OSAL_SEM_PostISR(_DRV_USART_OBJ(dObj, rxSemID));

} /* DRV_USART_TasksRX */
```

Mutex Operation

A mutex or mutual exclusion is used to protect a shared resource from access by multiple threads at the same time. A shared resource may be a common data structure in RAM or it may be a hardware peripheral. In either case a mutex can be used to ensure the integrity of the entire resource by only allowing one thread to access it at a time.

Description

The library must be written in such a way that before the shared resources is accessed the mutex has to be obtained. Once obtained the accesses should occur, and once complete the mutex should then be released. While no restrictions are enforced the sequence of operations between the lock and unlock should ideally take as few lines of code as possible to ensure good system performance.

The mutex may be implemented as a form of binary semaphore but an underlying RTOS will often add other features. It is normal to add the restriction that a mutex may only be unlocked from the thread that originally obtained the lock in the first place. The RTOS may also provide features to mitigate priority inversion problems (where a high priority thread blocks on a lower priority one holding a mutex) by providing priority inheritance allowing lower priority threads to be temporarily raised to complete and release a locked mutex.

```
/* perform operations on a shared data structure */
struct myDataStructure {
    uint16_t x;
    uint8_t y;
} myDataStructure;

...
OSAL_MUTEX_DECLARE(mutexDS);
OSAL_MUTEX_Create(&mutexDS);

...
/* wait 2 seconds to obtain the mutex */
if (OSAL_MUTEX_Lock(mutexDS, 2000) == OSAL_RESULT_TRUE)
{
    /* operate on the data structure */
    myDataStructure.x = 32;
    OSAL_MUTEX_Unlock(mutexDS);
}
```

Critical Section Operation

This section describes how critical sections are used.

Description

Critical sections are used to form sequences of code that must operate in an atomic manner. The interface allows for the possibility of two types of critical section.

- When the critical section is entered all interrupts on the microcontroller are disabled. This prevents the protected sequence of code from being interrupted and ensures the complete atomicity of the operation. This is denoted by the OSAL_CRIT_TYPE_HIGH value
- When the critical section is entered the RTOS scheduler is disabled. In this second case other threads are prevented from running however interrupts can still occur which allows any asynchronous events to still be received and for the temporal accuracy of the RTOS scheduler to be maintained. This is denoted by the OSAL_CRIT_TYPE_LOW value

Since the behavior in the two cases is different the type of critical section must be identified in both the call to enter and leave.

```
/* enter and leave a critical section disabling interrupts */
OSAL_CRIT_Enter(OSAL_CRIT_TYPE_HIGH);
/* perform an atomic sequence of code */
...
/* leave the critical section */
OSAL_CRIT_Leave(OSAL_CRIT_TYPE_HIGH);
```

The underlying RTOS may not support the second scenario, in which case the OSAL implementation will default to disabling all interrupts.

Memory Operation

This section describes the memory operation using the OSAL Library.

Description

The OSAL Library provides an interface to a memory allocation mechanism. The memory required for dynamic instantiation of variables is normally provided by allocating it from the heap. However the standard C library implementation of malloc and free are not considered thread safe and so OSAL specific functions must be used if MPLAB Harmony or the application requires dynamic memory during operation.

When operating without an underlying RTOS the OSAL memory allocators default to using standard malloc and free functions. However, when operating with an RTOS the calls will defer to the specific scheme used by the RTOS. This may involve multiple memory pools or it may simply involve adding a critical section around calls to malloc and free. It is left to the implementation to define the most appropriate scheme.

```
/* allocate a large buffer */
uint8_t* buffer;

buffer = OSAL_Malloc(8000);
if (buffer != NULL)
{
    ... manipulate the buffer
    /* free the buffer */
    OSAL_Free(buffer);
    buffer = NULL;
}
```

OSAL Operation

This section describes OSAL control features.

Description

When the OSAL is using an underlying RTOS it may be necessary to allow the RTOS to perform one-time initialization before any calls to it are made. For instance, the RTOS might implement multiple memory pools for managing queues and semaphores, and it must be given the chance to create these pools before any of the objects are created. For this reason the application program should call [OSAL_Initialize](#) early on and certainly before any MPLAB Harmony drivers or middleware is initialized (since these may also create OSAL objects at creation time).

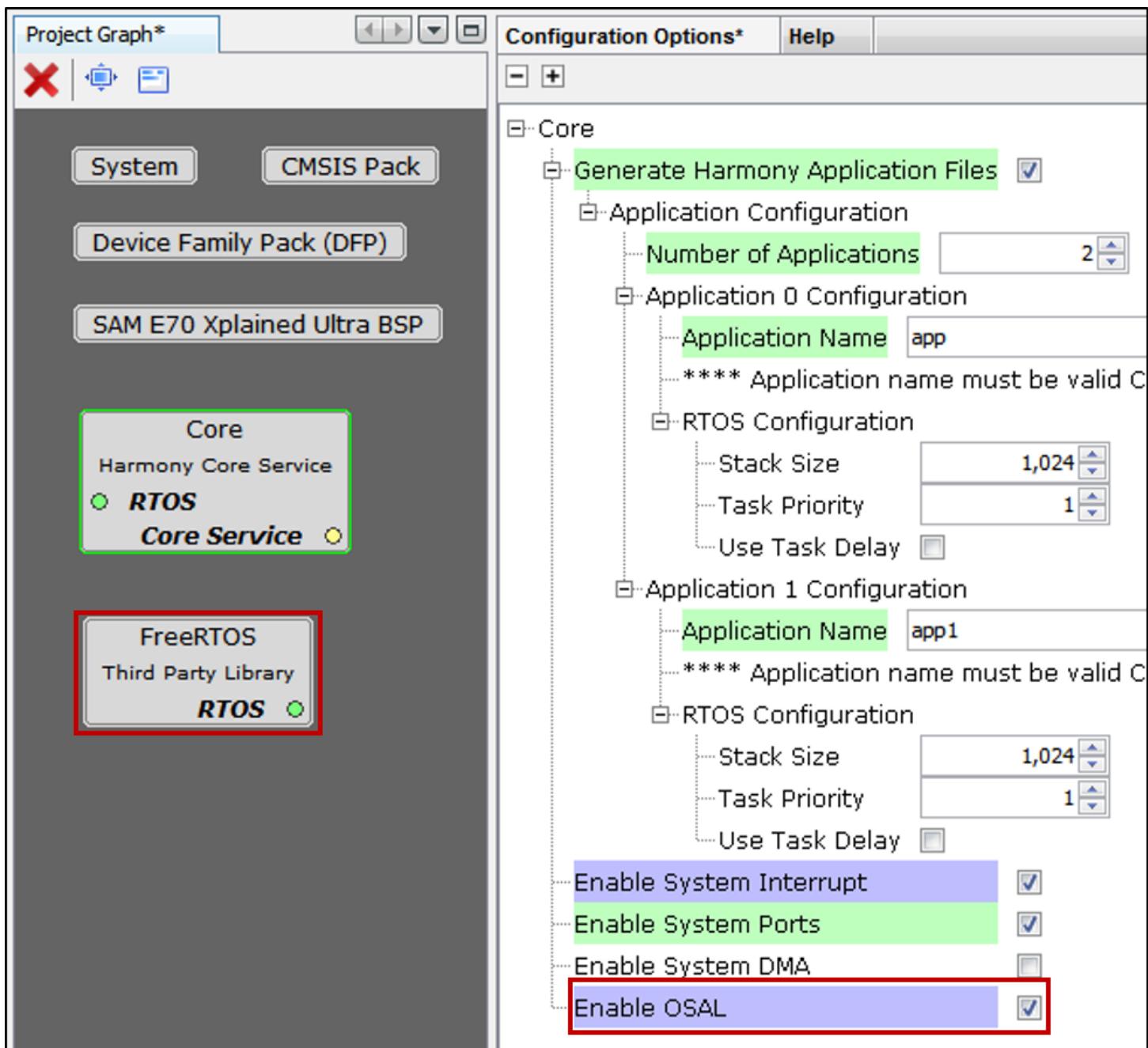
Once the OSAL is initialized and any other remaining parts of the system are configured correctly, the specific RTOS can be started.

Configuring the Library

This Section provides information on how to configure the OSAL component.

Description

OSAL can be configured via MHC. Below is the snapshot of the MHC configuration window for configuring the OSAL and a brief description of various configuration options.



OSAL can be enabled by user in two way:

1. User can manually enable OSAL which is part of Harmony Core component
2. OSAL will get auto enabled and set to "BareMetal" when Middleware or any Driver is instantiated. If any Third-Party Library i.e. RTOS is added then the respective OSAL Layer code will get generated. For instance if user instantiates FreeRTOS then OSAL is set to "FreeRTOS" and "osal_freertos.c" code will get generated during code generation.

Building the Library

This section lists the files that are available in the OSAL 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>/osal.

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 for

the selected RTOS.

Source File Name	Description
/osal.h	This file provides the interface definitions of the OSAL Library.

Required File(s)

All of the required files are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

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

Source File Name	Description
/src/osal.c	Source files added to the project if using the "BareMetal" OSAL basic implementation used when no RTOS is present.
/src/osal_impl_basic.h	
/src/osal_freertos.c	Source files added to the project if using the FreeRTOS implementation for compatibility with the current version 10.0.1 of the FreeRTOS operating system from Amazon.
/src/osal_freertos.h	

Optional File(s)

There are no optional files for the OSAL Library.

Library Interface

a) Semaphore Functions

	Name	Description
≡	OSAL_SEM_Create	Creates an OSAL Semaphore.
≡	OSAL_SEM_Delete	Deletes an OSAL Semaphore.
≡	OSAL_SEM_Pend	Waits on a semaphore. Returns true if the semaphore was obtained within the time limit.
≡	OSAL_SEM_Post	Posts a semaphore or increments a counting semaphore.
≡	OSAL_SEM_PostISR	Posts a semaphore or increments a counting semaphore from within an Interrupt Service Routine (ISR).
≡	OSAL_SEM_GetCount	Returns the current value of a counting semaphore.

b) Mutex Functions

	Name	Description
≡	OSAL_MUTEX_Create	Creates a mutex.
≡	OSAL_MUTEX_Delete	Deletes a mutex.
≡	OSAL_MUTEX_Lock	Locks a mutex.
≡	OSAL_MUTEX_Unlock	Unlocks a mutex.

c) Critical Section Functions

	Name	Description
≡	OSAL_CRIT_Enter	Enters a critical section with the specified severity level.
≡	OSAL_CRIT_Leave	Leaves a critical section with the specified severity level.

d) Memory Allocation Functions

	Name	Description
≡	OSAL_Malloc	Allocates memory using the OSAL default allocator.
≡	OSAL_Free	Deallocates a block of memory and return to the default pool.

e) OSAL Control Functions

	Name	Description
≡	OSAL_Initialize	Performs OSAL initialization.

	OSAL_Name	Obtains the name of the underlying RTOS.
---	---------------------------	--

f) Data Types and Constants

	Name	Description
	OSAL_CRIT_TYPE	Enumerated type representing the possible types of critical section.
	OSAL_RESULT	Enumerated type representing the general return value from OSAL functions.
	OSAL_SEM_TYPE	Enumerated type representing the possible types of semaphore.
	OSAL_SEM_DECLARE	Declares an OSAL semaphore.
	OSAL_MUTEX_DECLARE	Declares an OSAL mutex.
	_OSAL_H	This is macro _OSAL_H.

Description

This section describes the APIs of the OSAL Library. Refer to each section for a description.

a) Semaphore Functions

OSAL_SEM_Create Function

Creates an OSAL Semaphore.

File

[help_osal.h](#)

C

```
OSAL_RESULT OSAL_SEM_Create(OSAL_SEM_HANDLE_TYPE* semID, OSAL_SEM_TYPE type, uint8_t maxCount,
                            uint8_t initialCount);
```

Returns

- OSAL_RESULT_TRUE - Semaphore created
- OSAL_RESULT_FALSE - Semaphore creation failed
- semID - Updated with valid semaphore handle if call was successful

Description

This function creates an OSAL binary or counting semaphore. If OSAL_SEM_TYPE_BINARY is specified, the maxcount and initialCount values are ignored.

Remarks

None.

Preconditions

Semaphore must have been declared.

Example

```
OSAL_SEM_Create(&mySemID, OSAL_SEM_TYPE_COUNTING, 10, 5);
```

Parameters

Parameters	Description
semID	Pointer to the Semaphore ID
type	If OSAL_SEM_TYPE_BINARY, create a binary semaphore. If OSAL_SEM_TYPE_COUNTING, create a counting semaphore with the specified count values.

maxCount	Maximum value for a counting semaphore (ignored for a BINARY semaphore). This parameter is ignored for Express Logic ThreadX, SEGGER embOS, and Micrium/OS-III as these RTOS libraries do not support this parameter.
initialCount	Starting count value for the semaphore (ignored for a BINARY semaphore) This should be less than or equal to maxCount when used with a RTOS library that supports the maxCount parameter (i.e., FreeRTOS and OPENRTOS)

Function

`OSAL_RESULT OSAL_SEM_Create(OSAL_SEM_HANDLE_TYPE* semID, OSAL_SEM_TYPE type, uint8_t maxCount, uint8_t initialCount)`

OSAL_SEM_Delete Function

Deletes an OSAL Semaphore.

File

[help_osal.h](#)

C

`OSAL_RESULT OSAL_SEM_Delete(OSAL_SEM_HANDLE_TYPE* semID);`

Returns

- OSAL_RESULT_TRUE - Semaphore deleted
- OSAL_RESULT_FALSE - Semaphore deletion failed

Description

This function deletes an OSAL semaphore.

Remarks

None.

Preconditions

Semaphore must have been created.

Example

`OSAL_SEM_Delete(&mySemID);`

Parameters

Parameters	Description
semID	Pointer to the Semaphore ID

Function

`OSAL_RESULT OSAL_SEM_Delete(OSAL_SEM_HANDLE_TYPE* semID)`

OSAL_SEM_Pend Function

Waits on a semaphore. Returns true if the semaphore was obtained within the time limit.

File

[help_osal.h](#)

C

`OSAL_RESULT OSAL_SEM_Pend(OSAL_SEM_HANDLE_TYPE* semID, uint16_t waitMS);`

Returns

- OSAL_RESULT_TRUE - Semaphore obtained
- OSAL_RESULT_FALSE - Semaphore not obtained or time-out occurred

Description

This function is a blocking function call that waits (i.e., pends) on a semaphore. The function will return true if the semaphore has been obtained, or false if it was not available or the time limit was exceeded.

Remarks

None.

Preconditions

Semaphore must have been created.

Example

```
if (OSAL_SEM_Pend(&semUARTRX, 50) == OSAL_RESULT_TRUE)
{
    // character available
    c = DRV_USART_ReadByte(drvID);
    ...
}
else
{
    // character not available, resend prompt
    ...
}
```

Parameters

Parameters	Description
semID	Pointer to the Semaphore ID
waitMS	Time limit to wait in milliseconds: <ul style="list-style-type: none"> • 0 - do not wait • OSAL_WAIT_FOREVER - return only when semaphore is obtained • Other values - time-out delay

Function

[OSAL_RESULT OSAL_SEM_Pend\(OSAL_SEM_HANDLE_TYPE* semID, uint16_t waitMS\)](#)

OSAL_SEM_Post Function

Posts a semaphore or increments a counting semaphore.

File

[help_osal.h](#)

C

```
OSAL_RESULT OSAL_SEM_Post(OSAL_SEM_HANDLE_TYPE* semID);
```

Returns

- OSAL_RESULT_TRUE - Semaphore posted
- OSAL_RESULT_FALSE - Semaphore not posted

Description

This function posts a binary semaphore or increments a counting semaphore. The highest priority task currently blocked on the semaphore will be released and made ready to run.

Remarks

None.

Preconditions

Semaphore must have been created.

Example

```
OSAL_SEM_Post(&semSignal);
```

Parameters

Parameters	Description
semID	The semID

Function

[OSAL_SEM_Post](#)*ISR Function*

Posts a semaphore or increments a counting semaphore from within an Interrupt Service Routine (ISR).

File

[help_osal.h](#)

C

```
OSAL_RESULT OSAL_SEM_PostISR(OSAL_SEM_HANDLE_TYPE* semID);
```

Returns

- OSAL_RESULT_TRUE - Semaphore posted
- OSAL_RESULT_FALSE - Semaphore not posted

Description

This function posts a binary semaphore or increments a counting semaphore. The highest priority task currently blocked on the semaphore will be released and made ready to run. This form of the post function should be used inside an ISR.

Remarks

This version of the [OSAL_SEM_Post](#) function should be used if the program is, or may be, operating inside an ISR. The OSAL will take the necessary steps to ensure correct operation possibly disabling interrupts or entering a critical section. The exact requirements will depend upon the particular RTOS being used.

Preconditions

Semaphore must have been created.

Example

```
void __ISR(UART_2_VECTOR) _UART2RXHandler()
{
    char c;

    // read the character
    c = U2RXREG;
    // clear the interrupt flag
    IFS1bits.U2IF = 0;
    // post a semaphore indicating a character has been received
    OSAL_SEM_PostISR(&semSignal);

}
```

Parameters

Parameters	Description
semID	Pointer to the Semaphore ID

Function

`OSAL_RESULT OSAL_SEM_PostISR(OSAL_SEM_HANDLE_TYPE* semID)`

OSAL_SEM_GetCount Function

Returns the current value of a counting semaphore.

File

[help_osal.h](#)

C

```
uint8_t OSAL_SEM_GetCount(OSAL_SEM_HANDLE_TYPE* semID);
```

Returns

- 0 - Semaphore is unavailable
- 1-255 - Current value of the counting semaphore

Description

This function returns the current value of a counting semaphore. The value returned is assumed to be a single value ranging from 0-255.

Remarks

None.

Preconditions

Semaphore must have been created.

Example

```
uint8_t semCount;

semCount = OSAL_SEM_GetCount(semUART);

if (semCount > 0)
{
    // obtain the semaphore
    if (OSAL_SEM_Pend(&semUART) == OSAL_RESULT_TRUE)
    {
        // perform processing on the comm channel
        ...
    }
}
else
{
    // no comm channels available
    ...
}
```

Parameters

Parameters	Description
semID	Pointer to the Semaphore ID

Function

`uint8_t OSAL_SEM_GetCount(OSAL_SEM_HANDLE_TYPE* semID)`

b) Mutex Functions

OSAL_MUTEX_Create Function

Creates a mutex.

File

[help_osal.h](#)

C

```
OSAL_RESULT OSAL_MUTEX_Create(OSAL_MUTEX_HANDLE_TYPE* mutexID);
```

Returns

- OSAL_RESULT_TRUE - Mutex successfully created
- OSAL_RESULT_FALSE - Mutex failed to be created

Description

This function creates a mutex, allocating storage if required and placing the mutex handle into the passed parameter.

Remarks

None.

Preconditions

Mutex must have been declared.

Example

```
OSAL_MUTEX_HANDLE_TYPE mutexData;
OSAL_MUTEX_Create(&mutexData);
...
if (OSAL_MUTEX_Lock(&mutexData, 1000) == OSAL_RESULT_TRUE)
{
    // manipulate the shared data
    ...
}
```

Parameters

Parameters	Description
mutexID	Pointer to the mutex handle

Function

[OSAL_RESULT](#) OSAL_MUTEX_Create(OSAL_MUTEX_HANDLE_TYPE* mutexID)

OSAL_MUTEX_Delete Function

Deletes a mutex.

File

[help_osal.h](#)

C

```
OSAL_RESULT OSAL_MUTEX_Delete(OSAL_MUTEX_HANDLE_TYPE* mutexID);
```

Returns

- OSAL_RESULT_TRUE - Mutex successfully deleted
- OSAL_RESULT_FALSE - Mutex failed to be deleted

Description

This function deletes a mutex and frees associated storage if required.

Remarks

None.

Preconditions

None.

Example

```
OSAL_MUTEX_Delete(&mutexData);
```

Parameters

Parameters	Description
mutexID	Pointer to the mutex handle

Function

```
OSAL_RESULT OSAL_MUTEX_Delete(OSAL_MUTEX_HANDLE_TYPE* mutexID)
```

OSAL_MUTEX_Lock Function

Locks a mutex.

File

[help_osal.h](#)

C

```
OSAL_RESULT OSAL_MUTEX_Lock(OSAL_MUTEX_HANDLE_TYPE* mutexID, uint16_t waitMS);
```

Returns

- OSAL_RESULT_TRUE - Mutex successfully obtained
- OSAL_RESULT_FALSE - Mutex failed to be obtained or time-out occurred

Description

This function locks a mutex, waiting for the specified time-out. If it cannot be obtained or the time-out period elapses 'false' is returned.

Remarks

None.

Preconditions

Mutex must have been created.

Example

```
...
if (OSAL_MUTEX_Lock(&mutexData, 1000) == OSAL_RESULT_TRUE)
{
    // manipulate the shared data
    ...
    // unlock the mutex
    OSAL_MUTEX_Unlock(&mutexData);
}
```

Parameters

Parameters	Description
mutexID	Pointer to the mutex handle
waitMS	Time-out value in milliseconds: <ul style="list-style-type: none"> • 0, do not wait return immediately • OSAL_WAIT_FOREVER, wait until mutex is obtained before returning • Other values, time-out delay

Function

[OSAL_RESULT OSAL_MUTEX_Lock\(OSAL_MUTEX_HANDLE_TYPE* mutexID, uint16_t waitMS\)](#)

OSAL_MUTEX_Unlock Function

Unlocks a mutex.

File

[help_osal.h](#)

C

```
OSAL_RESULT OSAL_MUTEX_Unlock(OSAL_MUTEX_HANDLE_TYPE* mutexID);
```

Returns

- OSAL_RESULT_TRUE - Mutex released
- OSAL_RESULT_FALSE - Mutex failed to be released or error occurred

Description

This function unlocks a previously obtained mutex.

Remarks

None.

Preconditions

Mutex must have been created.

Example

```
...
if (OSAL_MUTEX_Lock(&mutexData, 1000) == OSAL_RESULT_TRUE)
{
    // manipulate the shared data
    ...

    // unlock the mutex
    OSAL_MUTEX_Unlock(&mutexData);
}
```

Parameters

Parameters	Description
mutexID	Pointer to the mutex handle

Function

[OSAL_RESULT OSAL_MUTEX_Unlock\(OSAL_MUTEX_HANDLE_TYPE* mutexID\)](#)

c) Critical Section Functions

OSAL_CRIT_Enter Function

Enters a critical section with the specified severity level.

File

[help_osal.h](#)

C

```
OSAL_CRITSECT_DATA_TYPE OSAL_CRIT_Enter(OSAL_CRIT_TYPE severity);
```

Returns

A data type of OSAL_CRITSECT_DATA_TYPE, this value represents the state of interrupts before entering the critical section.

Description

This function enters a critical section of code. It is assumed that the sequence of operations bounded by the enter and leave critical section operations is treated as one atomic sequence that will not be disturbed. This function should be paired with [OSAL_CRIT_Leave\(\)](#).

Remarks

The sequence of operations bounded by the OSAL_CRIT_Enter and [OSAL_CRIT_Leave](#) form a critical section. The severity level defines whether the RTOS should perform task locking or completely disable all interrupts.

Preconditions

None.

Example

```
OSAL_CRITSECT_DATA_TYPE IntState;
// prevent other tasks preempting this sequence of code
IntState = OSAL_CRIT_Enter(OSAL_CRIT_TYPE_HIGH);
// modify the peripheral
DRV_USART_Reinitialize( objUSART, &initData );
OSAL_CRIT_Leave(OSAL_CRIT_TYPE_HIGH, IntState);
```

Parameters

Parameters	Description
severity	OSAL_CRIT_TYPE_LOW, The RTOS should disable all other running tasks effectively locking the scheduling mechanism. OSAL_CRIT_TYPE_HIGH, The RTOS should disable all possible interrupts sources including the scheduler ensuring that the sequence of code operates without interruption. The state of interrupts are returned to the user before they are disabled.

Function

```
OSAL_CRITSECT_DATA_TYPE void OSAL_CRIT_Enter( OSAL_CRIT_TYPE severity)
```

OSAL_CRIT_Leave Function

Leaves a critical section with the specified severity level.

File

[help_osal.h](#)

C

```
void OSAL_CRIT_Leave(OSAL_CRIT_TYPE severity, OSAL_CRITSECT_DATA_TYPE status);
```

Returns

None.

Description

This function leaves a critical section of code. It is assumed that the sequence of operations bounded by the enter and leave critical section operations is treated as one atomic sequence that will not be disturbed. The severity should match the severity level used in the corresponding [OSAL_CRIT_Enter](#) call to ensure that the RTOS carries out the correct action.

Remarks

The sequence of operations bounded by the [OSAL_CRIT_Enter](#) and [OSAL_CRIT_Leave](#) form a critical section. The severity level defines whether the RTOS should perform task locking or completely disable all interrupts.

Preconditions

None.

Example

```
OSAL_CRITSECT_DATA_TYPE IntState;
// prevent other tasks preempting this sequence of code
intState = OSAL_CRIT_Enter(OSAL_CRIT_TYPE_LOW);
// modify the peripheral
DRV_USART_Reinitialize( objUSART, &initData );
OSAL_CRIT_Leave(OSAL_CRIT_TYPE_LOW, IntState);
```

Parameters

Parameters	Description
severity	OSAL_CRIT_TYPE_LOW, The scheduler will be unlocked, if no other nested calls to OSAL_CRIT_ENTER have been made. OSAL_CRIT_TYPE_HIGH, Interrupts are returned to the state passed into this function. The state should of been saved by an earlier call to OSAL_CRIT_Enter .
status	The value which will be used to set the state of global interrupts, if OSAL_CRIT_TYPE_HIGH is passed in.

Function

```
void OSAL_CRIT_Leave( OSAL_CRIT_TYPE severity, OSAL_CRITSECT_DATA_TYPE status)
```

d) Memory Allocation Functions

OSAL_Malloc Function

Allocates memory using the OSAL default allocator.

File

[help_osal.h](#)

C

```
void* OSAL_Malloc(size_t size);
```

Returns

Pointer to the block of allocated memory. NULL is returned if memory could not be allocated.

Description

This function allocates a block of memory from the default allocator from the underlying RTOS. If no RTOS is present, it defaults to malloc. Many operating systems incorporate their own memory allocation scheme, using pools, blocks or by wrapping the standard C library functions in a critical section. Since a MPLAB Harmony application may not know what target OS is being used

(if any), this function ensures that the correct thread-safe memory allocator will be used.

Remarks

None.

Preconditions

None.

Example

```
// create a working array
uint8_t* pData;

pData = OSAL_Malloc(32);
if (pData != NULL)
{
    ...
}
```

Parameters

Parameters	Description
size	Size of the requested memory block in bytes

Function

void* OSAL_Malloc(size_t size)

OSAL_Free Function

Deallocates a block of memory and return to the default pool.

File

help_osal.h

C

```
void OSAL_Free(void* pData);
```

Returns

None.

Description

This function deallocates memory and returns it to the default pool. In an RTOS-based application, the memory may have been allocated from multiple pools or simply from the heap. In non-RTOS applications, this function calls the C standard function free.

Remarks

None.

Preconditions

None.

Example

```
// create a working array
uint8_t* pData;

pData = OSAL_Malloc(32);
if (pData != NULL)
{
    ...
}

// deallocate the memory
OSAL_Free(pData);
```

```
// and prevent it accidentally being used again
pData = NULL;
}
```

Parameters

Parameters	Description
pData	Pointer to the memory block to be set free

Function

`void OSAL_Free(void* pData)`

e) OSAL Control Functions

OSAL_Initialize Function

Performs OSAL initialization.

File

[help_osal.h](#)

C

`OSAL_RESULT OSAL_Initialize();`

Returns

`OSAL_RESULT_TRUE` - Initialization completed successfully.

Description

This function performs OSAL initialization. This function should be called near the start of main in an application that will use an underlying RTOS. This permits the RTOS to perform any one time initialization before the application attempts to create drivers or other items that may use the RTOS. Typical actions performed by `OSAL_Initialize` would be to allocate and prepare any memory pools for later use.

Remarks

None.

Preconditions

None.

Example

```
int main()
{
    OSAL_Initialize();

    App_Init();
    OSAL_Start();
}
```

Function

[OSAL_RESULT OSAL_Initialize\(\)](#)

OSAL_Name Function

Obtains the name of the underlying RTOS.

File[help_osal.h](#)**C**

```
const char* OSAL_Name();
```

Returns

const char* - Name of the underlying RTOS or NULL

Description

This function returns a const char* to the textual name of the RTOS. The name is a NULL terminated string.

Remarks

None.

Preconditions

None.

Example

```
// get the RTOS name
const char* sName;

sName = OSAL_Name();
sprintf(buff, "RTOS: %s", sName);
```

Function

```
const char* OSAL_Name()
```

f) Data Types and Constants**OSAL_CRIT_TYPE Enumeration**

Enumerated type representing the possible types of critical section.

File[help_osal.h](#)**C**

```
enum OSAL_CRIT_TYPE {
    OSAL_CRIT_TYPE_LOW,
    OSAL_CRIT_TYPE_HIGH
};
```

Description

OSAL Critical Type

This enum represents possible critical section types.

OSAL_CRIT_TYPE_LOW - Low priority critical section, can be formed by locking the scheduler (if supported by RTOS)

OSAL_CRIT_TYPE_HIGH - High priority critical section, will be formed by disabling all interrupts.

Remarks

Critical section types.

OSAL_RESULT Enumeration

Enumerated type representing the general return value from OSAL functions.

File

[help_osal.h](#)

C

```
enum OSAL_RESULT {
    OSAL_RESULT_NOT_IMPLEMENTED = -1,
    OSAL_RESULT_FALSE = 0,
    OSAL_RESULT_TRUE = 1
};
```

Description

OSAL Result type

This enum represents possible return types from OSAL functions.

Remarks

These enum values are the possible return values from OSAL functions where a standard success/fail type response is required.

The majority of OSAL functions will return this type with a few exceptions.

OSAL_SEM_TYPE Enumeration

Enumerated type representing the possible types of semaphore.

File

[help_osal.h](#)

C

```
enum OSAL_SEM_TYPE {
    OSAL_SEM_TYPE_BINARY,
    OSAL_SEM_TYPE_COUNTING
};
```

Description

OSAL Semaphore Type

This enum represents possible semaphore types.

OSAL_SEM_TYPE_BINARY - Simple binary type that can be taken once

OSAL_SEM_TYPE_COUNTING - Complex type that can be taken set number of times defined at creation time

Remarks

Binary and counting semaphore type.

OSAL_SEM_DECLARE Macro

Declares an OSAL semaphore.

File

[help_osal.h](#)

C

```
#define OSAL_SEM_DECLARE(semID) OSAL_SEM_HANDLE_TYPE semID
```

Description

`OSAL_SEM_Declare(semID)`

This function declares a data item of type `OSAL_SEM_HANDLE_TYPE`.

Remarks

None.

OSAL_MUTEX_DECLARE Macro

Declares an OSAL mutex.

File

[help_osal.h](#)

C

```
#define OSAL_MUTEX_DECLARE(mutexID) OSAL_MUTEX_HANDLE_TYPE mutexID
```

Description

`OSAL_MUTEX_Declare(mutexID)`

This function declares a data item of type `OSAL_MUTEX_HANDLE_TYPE`.

Remarks

None.

// DOM-IGNORE-BEGIN

Place the definition of the `OSAL_MUTEX_Declare` macro inside each specific implementation file. Not all implementation files use the same definition. The type for each individual RTOS is different, and to handle this properly in the code, each implementation must define its own declaration. ****REMOVE THIS NOTE AND THE DOM STATEMENTS IF USING THIS API IN A FILE THAT IS INCLUDED IN THE MPLAB HARMONY INSTALLATION**** // DOM-IGNORE-END

_OSAL_H Macro

File

[help_osal.h](#)

C

```
#define _OSAL_H
```

Description

This is macro `_OSAL_H`.

Files

Files

Name	Description
osal.h	Common interface definitions for the Operating System Abstraction Layer (OSAL).
help_osal.h	Common interface definitions for the Operating System Abstraction Layer (OSAL).

Description

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

osal.h

Common interface definitions for the Operating System Abstraction Layer (OSAL).

Description

Descriptive File Name: Operating System Abstraction Layer

This file defines the common interface to the Operating System Abstraction Layer. It defines the common types used by the OSAL and defines the function prototypes. Depending upon the OSAL mode, a support level specific implementation file is included by this file to give the required level of compatibility. The available support levels include, OSAL_USE_NONE, OSAL_USE_BASIC, and OSAL_USE_RTOS.

File Name

osal.h

Company

Microchip Technology Inc.

help_osal.h

Common interface definitions for the Operating System Abstraction Layer (OSAL).

Enumerations

	Name	Description
⌚	OSAL_CRIT_TYPE	Enumerated type representing the possible types of critical section.
⌚	OSAL_RESULT	Enumerated type representing the general return value from OSAL functions.
⌚	OSAL_SEM_TYPE	Enumerated type representing the possible types of semaphore.

Functions

	Name	Description
⌚	OSAL_CRIT_Enter	Enters a critical section with the specified severity level.
⌚	OSAL_CRIT_Leave	Leaves a critical section with the specified severity level.
⌚	OSAL_Free	Deallocates a block of memory and return to the default pool.
⌚	OSAL_Initialize	Performs OSAL initialization.
⌚	OSAL_Malloc	Allocates memory using the OSAL default allocator.
⌚	OSAL_MUTEX_Create	Creates a mutex.
⌚	OSAL_MUTEX_Delete	Deletes a mutex.
⌚	OSAL_MUTEX_Lock	Locks a mutex.
⌚	OSAL_MUTEX_Unlock	Unlocks a mutex.
⌚	OSAL_Name	Obtains the name of the underlying RTOS.
⌚	OSAL_SEM_Create	Creates an OSAL Semaphore.
⌚	OSAL_SEM_Delete	Deletes an OSAL Semaphore.
⌚	OSAL_SEM_GetCount	Returns the current value of a counting semaphore.
⌚	OSAL_SEM_Pend	Waits on a semaphore. Returns true if the semaphore was obtained within the time limit.
⌚	OSAL_SEM_Post	Posts a semaphore or increments a counting semaphore.
⌚	OSAL_SEM_PostISR	Posts a semaphore or increments a counting semaphore from within an Interrupt Service Routine (ISR).

Macros

	Name	Description
	_OSAL_H	This is macro _OSAL_H.

	OSAL_MUTEX_DECLARE	Declares an OSAL mutex.
	OSAL_SEM_DECLARE	Declares an OSAL semaphore.

Description

Operating System Abstraction Layer

This file defines the common interface to the Operating System Abstraction Layer. It defines the common types used by the OSAL and defines the function prototypes. Depending upon the OSAL mode, a support level specific implementation file is included by this file to give the required level of compatibility. The available support levels include, OSAL_USE_NONE, OSAL_USE_BASIC, and OSAL_USE_RTOS.

File Name

[osal.h](#)

Company

Microchip Technology Inc.

Applications Help

This topic provides help for the Drivers, System Service, Middleware applications that are available in the Microchip 32 bit Harmony Core Package (CORE).

Refer to each applications help section

Driver Applications

This section provides help for the Driver applications.

AT24 Driver Applications

This section provides help for the AT24 driver applications.

at24_eeprom_read_write

This example application shows how to use the AT24 driver to perform operations on AT24 series of EEPROM.

Description

This example uses the AT24 driver to communicate with I2C based AT24 series EEPROM's to perform write and read operations in both Bare-Metal and RTOS environment.

The application communicates with following EEPROM's based on project configurations selected.

- External AT24CM02 EEPROM
- On-Board AT24MAC402

Building the Application

This section provides information on how to build an application using MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core\apps\driver\at24\at24_eeprom_read_write\firmware
------------------	---

To build the application, refer the following table and open the appropriate project file in MPLAB X IDE:

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_d20_xpro.X	SAM D20 Xplained Pro Evaluation Kit
sam_d21_xpro.X	SAM D21 Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult.X	SAM V71 Xplained Ultra Evaluation Kit
sam_v71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- AT24 Driver

The following components used may vary based on the project configuration selected:

- TWIHS or SERCOM peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Settings:

- Configure Pins for TWIHS or SERCOM peripheral library in Pin Settings based on project configuration selected
- Configure EEPROM write protect pin in Pin Settings

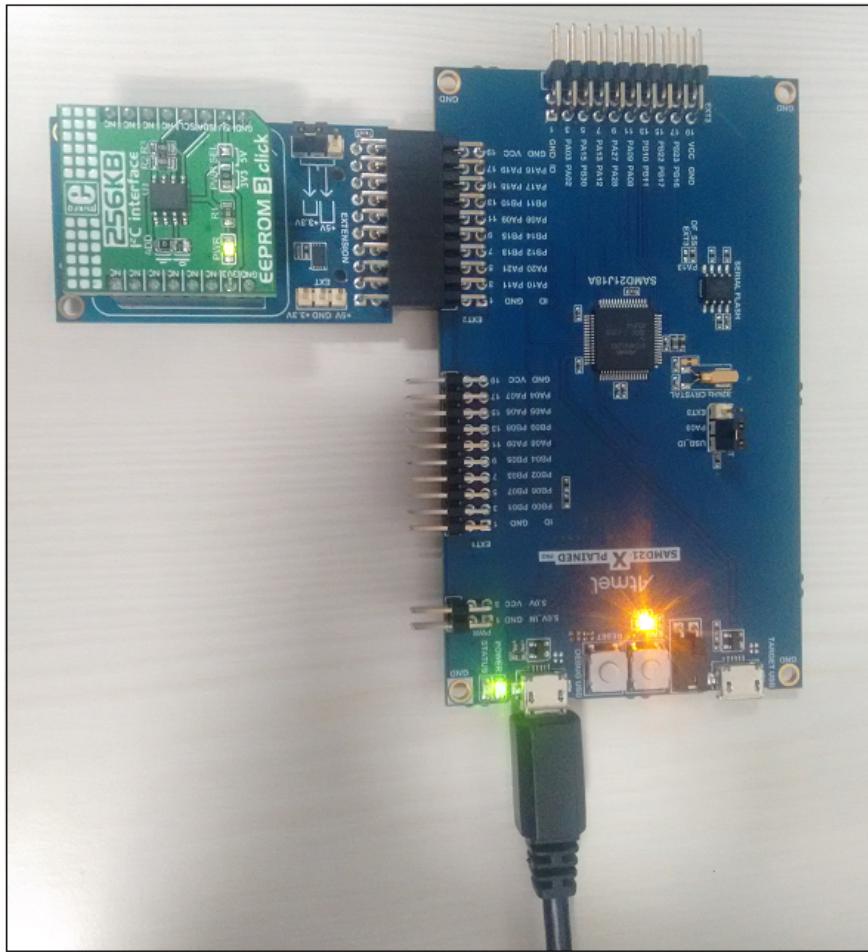
Hardware Setup

This section describes how to configure the supported hardware.

Description

Common Hardware Setup:

- Install an [EEPROM 3 click board](#) on to the [microBUS Xplained Pro](#) board
1. **Project sam_c21n_xpro.X.**
 - **Hardware Used:**
 - [SAM C21N Xplained Pro Evaluation Kit](#)
 - **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT2 header of the [SAM C21N Xplained Pro Evaluation Kit](#)
2. **Project sam_d20_xpro.X.**
 - **Hardware Used:**
 - [SAM D20 Xplained Pro Evaluation Kit](#)
 - **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT2 header of the [SAM D20 Xplained Pro Evaluation Kit](#)
3. **Project sam_d21_xpro.X.**
 - **Hardware Used:**
 - [SAM D21 Xplained Pro Evaluation Kit](#)
 - **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT2 header of the [SAM D21 Xplained Pro Evaluation Kit](#)



4. Project `sam_e70_xult.X`.

5. Project `sam_e70_xult_freertos.X`.

- **Hardware Used:**

- SAM E70 Xplained Ultra Evaluation Kit

- **Hardware Setup:**

- No special hardware setup required, SAM E70 Xplained Ultra Evaluation Kit has an On-Board EEPROM

6. Project `sam_v71_xult.X`.

7. Project `sam_v71_xult_freertos.X`.

- **Hardware Used:**

- [SAM V71 Xplained Ultra Evaluation Kit](#)

- **Hardware Setup:**

- No special hardware setup required, [SAM V71 Xplained Ultra Evaluation Kit](#) has an On-Board EEPROM

Running the Application

This section provides information on how to run the application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.

- The LED is turned ON when the value read from EEPROM matches with the written data
- The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM D20 Xplained Pro Evaluation Kit	LED 0
SAM D21 Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 0

AT25 Driver Applications

This section provides help for the AT25 driver applications.

at25_eeprom_read_write

This example application shows how to use the AT25 driver to perform operations on AT25 series of EEPROM.

Description

This example uses the AT25 driver to communicate with SPI based AT25 series external EEPROM's to perform write and read operations in both Bare-Metal and RTOS environment.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/at25/at25_eeprom_read_write/firmware

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_d20_xpro.X	SAM D20 Xplained Pro Evaluation Kit
sam_d21_xpro.X	SAM D21 Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freetos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult.X	SAM V71 Xplained Ultra Evaluation Kit
sam_v71_xult_freetos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- AT25 Driver

The following components used may vary based on the project configuration selected:

- SPI or SERCOM peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Settings:

- Configure Pins for AT25 Chip Select, Hold Pin and Write Protect Pin in Pin Settings

Hardware Setup

This section describes how to configure the supported hardware.

Description

Common Hardware Setup:

- Install an [EEPROM 4 Click](#) board on to the [microBUS Xplained Pro](#) board

1. Project sam_c21n_xpro.X.

- **Hardware Used:**
 - [SAM C21n Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Connect [microBUS Xplained Pro](#) board to the EXT2 header of the [SAM C21n Xplained Pro Evaluation Kit](#)

2. Project sam_d20_xpro.X.

- **Hardware Used:**
 - [SAM D20 Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT2 header of the [SAM D20 Xplained Pro Evaluation Kit](#)

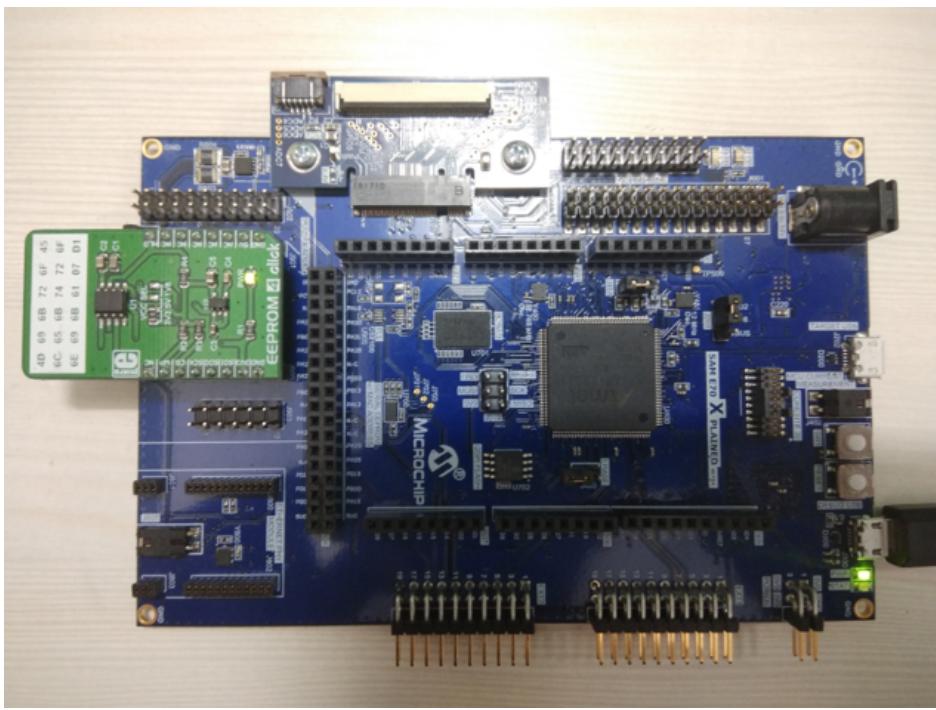
3. Project sam_d21_xpro.X.

- **Hardware Used:**
 - [SAM D21 Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT2 header of the [SAM D21 Xplained Pro Evaluation Kit](#)

4. Project sam_e70_xult.X.

5. Project sam_e70_xult_freertos.X.

- **Hardware Used:**
 - [SAM E70 Xplained Ultra Evaluation Kit](#)
- **Hardware Setup:**
 - Install [EEPROM 4 Click](#) board to the click slot provided on the SAM E70 Xplained Ultra Evaluation Kit as shown below:



6. Project sam_v71_xult.X.

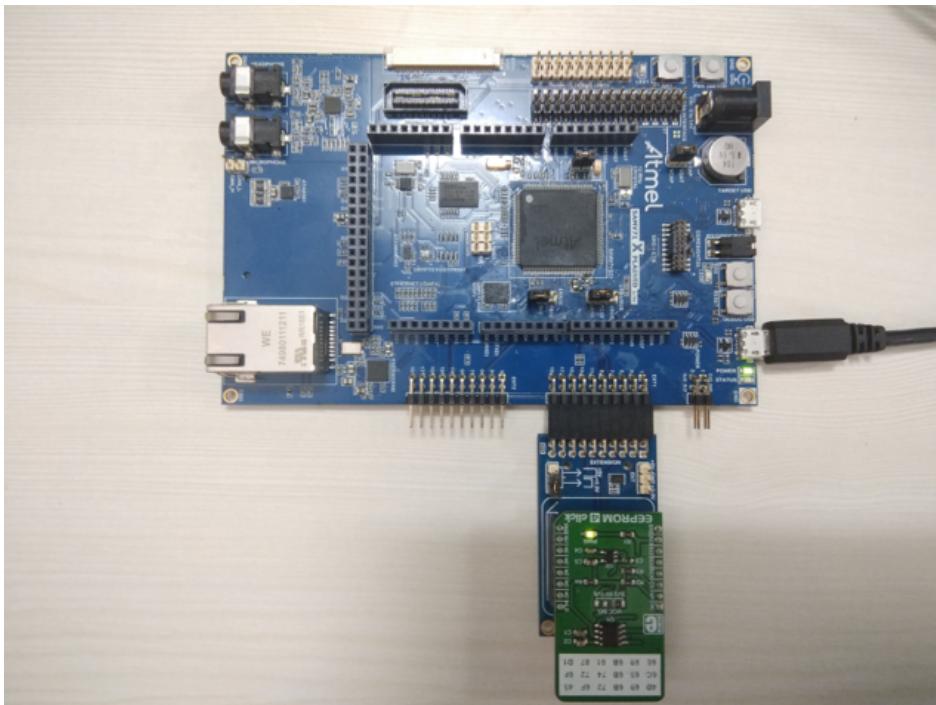
7. Project sam_v71_xult_freertos.X.

- **Hardware Used:**

- [SAM V71 Xplained Ultra Evaluation Kit](#)

- **Hardware Setup:**

- Connect [microBUS Xplained Pro](#) board to the EXT1 header of the [SAM V71 Xplained Ultra Evaluation Kit](#) as shown below.



Running the Application

This section provides information on how to run the application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED is turned ON when the value read from EEPROM matches with the written data

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM D20 Xplained Pro Evaluation Kit	LED 0
SAM D21 Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 2
SAM V71 Xplained Ultra Evaluation Kit	LED 0

Note:

Make sure a jumper is connected between **LED2** and PB08 on J204 header of SAM E70 Xplained Ultra Evaluation Kit.

I2C Driver Applications

Async

Name	Description
i2c_eeprom	This example application shows how to use the I2C driver in asynchronous mode to perform operations on the EEPROM.
i2c_multi_slave	This example application shows how to use the I2C driver in asynchronous mode to communicate with the external EEPROM and temperature sensor.

Sync

Name	Description
i2c_eeprom	This example application shows how to use the I2C driver in synchronous mode to perform operations on the EEPROM.
i2c_multi_slave	This example application shows how to use the I2C driver in synchronous mode to communicate with the external EEPROM and temperature sensor.

Description

This section provides help for the I2C driver applications.

Async

i2c_eeprom

This example application shows how to use the I2C driver in asynchronous mode to perform operations on the EEPROM.

Description

This example uses the I2C driver in asynchronous mode to communicate with the EEPROM to perform write and read operations in both Bare-Metal and RTOS environment.

The application communicates with the following EEPROM's based on the project configurations selected.

- External AT24CM02 EEPROM

- On-Board AT24MAC402

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/i2c/async/i2c_eeprom/firmware
------------------	--

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult.X	SAM V71 Xplained Ultra Evaluation Kit
sam_v71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- I2C Driver (Asynchronous mode)

The following components used may vary based on the project configuration selected:

- TWIHS or SERCOM peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Settings:

- Configure Pins for TWIHS or SERCOM peripheral library in Pin Settings based on project configuration selected
- Configure EEPROM write protect pin in Pin Settings

Hardware Setup

This section describes how to configure the supported hardware.

Description

Common Hardware Setup:

- Install an [EEPROM 3 click board](#) on to the [microBUS Xplained Pro](#) board

1. Project sam_c21n_xpro.X.

- **Hardware Used:**

- [SAM C21N Xplained Pro Evaluation Kit](#)
 - **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT2 header of the [SAM C21N Xplained Pro Evaluation Kit](#)
2. **Project sam_e70_xult.X.**
3. **Project sam_e70_xult_freertos.X.**
- **Hardware Used:**
 - SAM E70 Xplained Ultra Evaluation Kit
 - **Hardware Setup:**
 - No special hardware setup required, SAM E70 Xplained Ultra Evaluation Kit has an On-Board EEPROM
4. **Project sam_v71_xult.X.**
5. **Project sam_v71_xult_freertos.X.**
- **Hardware Used:**
 - [SAM V71 Xplained Ultra Evaluation Kit](#)
 - **Hardware Setup:**
 - No special hardware setup required, [SAM V71 Xplained Ultra Evaluation Kit](#) has on-board EEPROM

Running the Application

This section provides information on how to run the application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED is turned ON when the value read from EEPROM matches with the written data

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 1

i2c_multi_slave

This example application shows how to use the I2C driver in asynchronous mode to communicate with the external EEPROM and temperature sensor.

Description

The application demonstrates the multi-client feature of the I2C driver in asynchronous mode. The application uses a [IO1 Xplained Pro Extension Kit](#) that has a temperature sensor and an EEPROM interfaced on the same I2C bus

Two application tasks are created which act as clients to the same instance of the I2C driver:

1. APP_I2C_TEMP_SENSOR_Tasks():
 - Reads temperature for every 1 second and notifies the EEPROM client.
 - The Time System Service is used to generate a callback for every 1 second
2. APP_I2C_EEPROM_Tasks():
 - Writes temperature values to the EEPROM once the temperature read notification is received
 - Reads the temperature data back from the EEPROM and compares with the value written

- If the temperature value matches it displays the same on the console

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/i2c/async/i2c_multi_slave/firmware
-------------------------	--

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_d20_xpro.X	SAM D20 Xplained Pro Evaluation Kit
sam_d21_xpro.X	SAM D21 Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult.X	SAM V71 Xplained Ultra Evaluation Kit
sam_v71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- I2C Driver (Asynchronous mode). The Number of Clients is set to 2
- Time System Service
- TC peripheral library
- Standard Input Output (STDIO)

The following components used may vary based on the project configuration selected:

- TWIHS or SERCOM peripheral library
- USART or SERCOM peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Settings:

- Configure Pins for TWIHS or SERCOM peripheral library in Pin Settings based on project configuration selected
- Configure Pins for USART or SERCOM peripheral library in Pin Settings based on project configuration selected

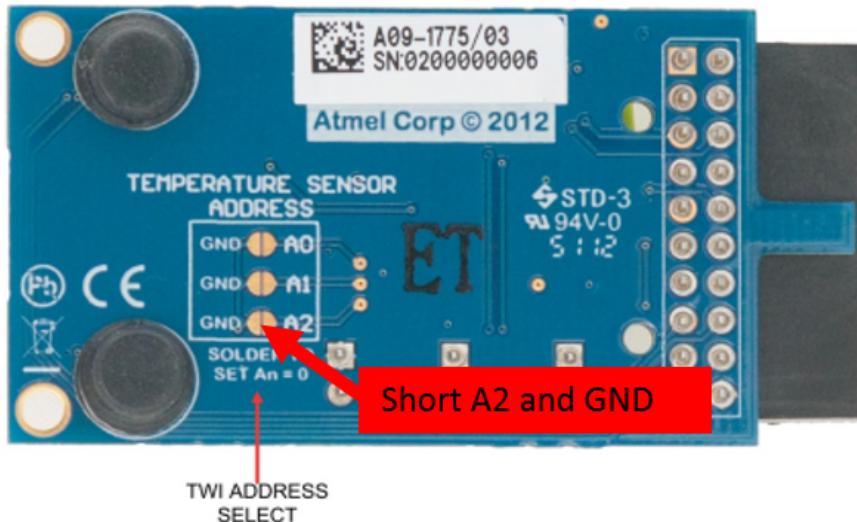
Hardware Setup

This section describes how to configure the supported hardware.

Description

Common Hardware Setup:

- IO1 Xplained Pro Extension Kit Address select:



- The A2 address line of TWI must be soldered to GND. This is done to modify the address of the the EEPROM on [IO1 Xplained Pro Extension Kit](#).
- The modification changes the address of the temperature sensor to **0x4B** and the EEPROM to **0x50**.

1. Project `sam_c21n_xpro.X`.

- **Hardware Used:**
 - [SAM C21N Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [IO1 Xplained Pro Extension Kit](#) to the EXT2 header of the [SAM C21N Xplained Pro Evaluation Kit](#)

2. Project `sam_d20_xpro.X`.

- **Hardware Used:**
 - [SAM D20 Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [IO1 Xplained Pro Extension Kit](#) to the EXT2 header of the [SAM D20 Xplained Pro Evaluation Kit](#)

3. Project `sam_d21_xpro.X`.

- **Hardware Used:**
 - [SAM D21 Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [IO1 Xplained Pro Extension Kit](#) to the EXT2 header of the [SAM D21 Xplained Pro Evaluation Kit](#)

4. Project `sam_e70_xult.X`.

5. Project `sam_e70_xult_freertos.X`.

- **Hardware Used:**
 - [SAM E70 Xplained Ultra Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [IO1 Xplained Pro Extension Kit](#) to the EXT1 header of the [SAM E70 Xplained Ultra Evaluation Kit](#)

6. Project `sam_v71_xult.X`.

7. Project `sam_v71_xult_freertos.X`.

- **Hardware Used:**

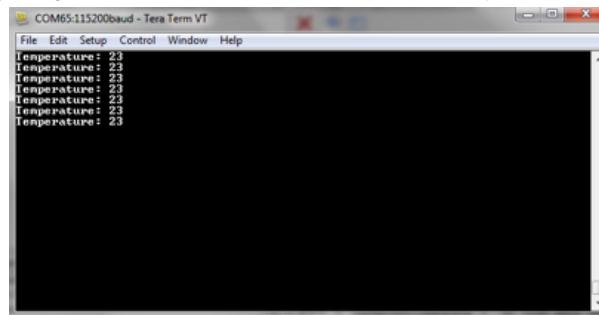
- [SAM V71 Xplained Ultra Evaluation Kit](#)
 - **Hardware Setup:**
 - Connect the [IO1 Xplained Pro Extension Kit](#) to the EXT1 header of the [SAM V71 Xplained Ultra Evaluation Kit](#)

Running the Application

This section provides information on how to run the application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
 2. Open Terminal application (Ex.:Tera term) on the computer.
 - Configure the DEBUG port settings as follows:
 - Baud : 115200
 - Data : 8 Bits
 - Parity : None
 - Stop : 1 Bit
 - Flow Control : None
 3. Build and program the application using the MPLAB X IDE.
 4. Observe the temperature values getting printed on the terminal application every 1 second as shown below:



Sync

i2c_eeprom

This example application shows how to use the I2C driver in synchronous mode to perform operations on the EEPROM.

Description

This example uses the I2C driver in synchronous mode to communicate with the EEPROM to perform write and read operations in RTOS environment.

The application communicates with the following EEPROM's based on the project configurations selected.

- External AT24CM02 EEPROM
 - On-Board AT24MAC402

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/i2c/sync/i2c_eeprom/firmware
------------------	---

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro_freertos.X	SAM C21N Xplained Pro Evaluation Kit + FreeRTOS
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- I2C Driver (Synchronous mode)
- FreeRTOS

The following components used may vary based on the project configuration selected:

- TWIHS or SERCOM peripheral library
- Board Support Package (BSP)

Other MHC Settings:

- Configure Pins for TWIHS or SERCOM peripheral library in Pin Settings based on project configuration selected
- Configure EEPROM write protect pin in Pin Settings

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_c21n_xpro_freertos.X.

- **Hardware Used:**
 - SAM C21N Xplained Pro Evaluation Kit
 - EEPROM 3 click board
 - mikroBUS Xplained Pro board
- **Hardware Setup:**
 - Connect the [EEPROM 3 click](#) board to the [mikroBUS Xplained Pro](#) board
 - Connect the [mikroBUS Xplained Pro](#) board to the EXT2 header of the [SAM C21N Xplained Pro Evaluation Kit](#)

2. Project sam_e70_xult_freertos.X.

- **Hardware Used:**
 - SAM E70 Xplained Ultra Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup required, SAM E70 Xplained Ultra Evaluation Kit has on-board EEPROM

Running the Application

This section provides information on how to run the application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED is turned ON when the value read from EEPROM matches with the written data

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1

i2c_multi_slave

This example application shows how to use the I2C driver in synchronous mode to communicate with the external EEPROM and temperature sensor.

Description

The application demonstrates the multi-client feature of the I2C driver in asynchronous mode. The application uses a [IO1 Xplained Pro Extension Kit](#) that has a temperature sensor and an EEPROM interfaced on the same I2C bus.

Two application threads are created which act as clients to the same instance of the I2C driver:

1. APP_I2C_TEMP_SENSOR_Tasks():
 - Remains in blocked state for 1 second.
 - Once unblocked, reads the temperature from the temperature sensor and posts a semaphore thereby unblocking the second (EEPROM) thread
1. APP_I2C_EEPROM_Tasks():
 - Remains blocked on a semaphore until a new temperature data is available
 - Once unblocked, writes the temperature data to the EEPROM
 - Reads the temperature value from the EEPROM and compares with the value written
 - If the temperature value matches it displays the same on the console.

Note:

Temperature data is written to the same EEPROM memory location.

Two application tasks are created

1. APP_I2C_TEMP_SENSOR_Tasks():
 - Reads temperature for every 1 second
 - The Time System Service is used to generate a callback for every 100 milliseconds
2. APP_I2C_EEPROM_Tasks():
 - Writes temperature values to the EEPROM
 - Reads and compares the value written

- If the temperature value matches it displays the same on the console

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/i2c/sync/i2c_multi_slave/firmware
------------------	--

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro_freertos.X	SAM C21N Xplained Pro Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- I2C Driver (Synchronous mode). The Number of Clients is set to 2
- Standard Input Output (STDIO)
- FreeRTOS

The following components used may vary based on the project configuration selected:

- TWIHS or SERCOM peripheral library
- USART or SERCOM peripheral library
- Board Support Package (BSP)

Other MHC Settings:

- Configure Pins for TWIHS or SERCOM peripheral library in Pin Settings based on project configuration selected
- Configure Pins for USART or SERCOM peripheral library in Pin Settings based on project configuration selected

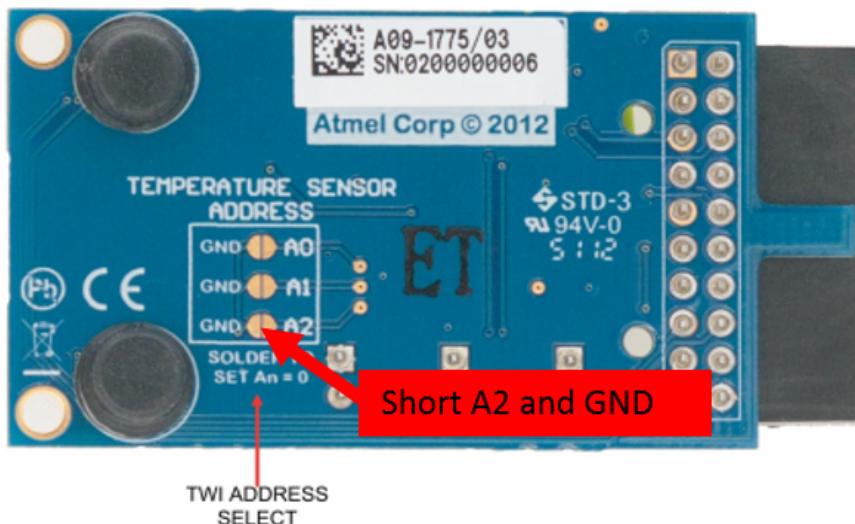
Hardware Setup

This section describes how to configure the supported hardware.

Description

Common Hardware Setup:

- [IO1 Xplained Pro Extension Kit](#) Address select:



- The A2 address line of TWI must be soldered to GND. This is done to modify the address of the the EEPROM on [IO1 Xplained Pro Extension Kit](#).
- The modification changes the address of the temperature sensor to **0x4B** and the EEPROM to **0x50**.

1. Project `sam_c21n_xpro_freertos.X`.

- **Hardware Used:**
 - [SAM C21N Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [IO1 Xplained Pro Extension Kit](#) to the EXT2 header of the [SAM C21N Xplained Pro Evaluation Kit](#)

2. Project `sam_e70_xult_freertos.X`.

- **Hardware Used:**
 - [SAM E70 Xplained Ultra Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [IO1 Xplained Pro Extension Kit](#) to the EXT1 header of the [SAM E70 Xplained Ultra Evaluation Kit](#)

3. Project `sam_v71_xult_freertos.X`.

- **Hardware Used:**
 - [SAM V71 Xplained Ultra Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [IO1 Xplained Pro Extension Kit](#) to the EXT1 header of the [SAM V71 Xplained Ultra Evaluation Kit](#)

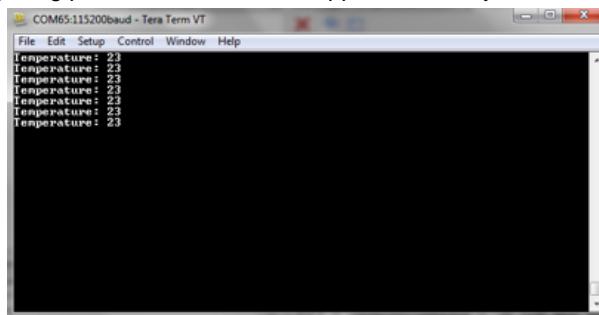
Running the Application

This section provides information on how to run the application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Open Terminal application (Ex.:Tera term) on the computer.
 - Configure the DEBUG port settings as follows:
 - Baud : 115200
 - Data : 8 Bits
 - Parity : None
 - Stop : 1 Bit
 - Flow Control : None
3. Build and program the application using the MPLAB X IDE.

4. Observe the temperature values getting printed on the terminal application every 1 second as shown below:



Memory Driver Applications

Async

Name	Description
nvm_sst26_read_write	This example application shows how to use the Memory driver in asynchronous mode to perform block operations on the NVM and the SST26 media's.

Sync

Name	Description
nvm_sst26_read_write	This example application shows how to use the Memory driver in synchronous mode to perform block operations on the NVM and the SST26 media's.

Description

This section provides help for the Memory driver applications.

Async

[nvm_sst26_read_write](#)

This example application shows how to use the Memory driver in asynchronous mode to perform block operations on the NVM and the SST26 media's.

Description

This application uses multi instances of the Memory driver to communicate with the NVM and the SST26 QSPI Flash memories in asynchronous mode of operation in both RTOS and Bare-Metal environment.

It Performs Block Erase/Write/Read operations on both the Media's.

The application consists of 5 tasks which are called through the SYS_Tasks() routine in Bare-Metal environment and though RTOS thread context in RTOS environment.

1. DRV_MEMORY_0_Tasks() : Manages the state machine of the Memory driver instance 0
2. DRV_MEMORY_1_Tasks() : Manages the state machine of the Memory driver instance 1
3. APP_SST26_Tasks() : Performs operations on the SST26 QSPI Flash memory.
4. APP_NVM_Tasks() : Performs operations on the NVM.
5. APP_MONITOR_Tasks(): Monitors the state of above two Tasks.

Building The Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/memory/async/nvm_sst26_read_write/firmware
-------------------------	--

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- Memory Driver 2 Instances
 - Instance 0 connected to the SST26 Driver
 - Instance 1 connected to the EFC peripheral library
- SST26 Driver
- QSPI peripheral library

The following components used may vary based on the project configuration selected:

- EFC peripheral library.
- Board Support Package (BSP)
- FreeRTOS

Other MHC configurations:

- Enable Memory Protection Unit (MPU) for the QSPI Flash memory region
- Enable Memory Protection Unit (MPU) for reserving Internal Flash region for the NVM operations

Hardware Setup

This section describes how to configure the supported hardware.

Description

- Project sam_e70_xult.X.
- Project sam_e70_xult_freertos.X.
 - Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
 - Hardware Setup**
 - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED is turned ON when the data read from each media matches with the data written in them.

The following table provides the LED name:

Kit Name	LED Name
SAM E70 Xplained Ultra Evaluation Kit	LED 1

Sync

nvm_sst26_read_write

This example application shows how to use the Memory driver in synchronous mode to perform block operations on the NVM and the SST26 media's.

Description

This application uses multi instances of the Memory driver to communicate with the NVM and the SST26 QSPI Flash memories in synchronous mode of operation in RTOS Environment.

It Performs Block Erase/Write/Read operations on both the Media's.

The application consists of 3 tasks which are called in the RTOS thread context:

1. APP_SST26_Tasks() : Performs operations on the SST26 QSPI Flash Memory
2. APP_NVM_Tasks() : Performs operations on the NVM
3. APP_MONITOR_Tasks(): Monitors the state of above two Tasks.

Building The Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/memory/sync/nvm_sst26_read_write/firmware
------------------	--

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- Memory Driver 2 Instances
 - Instance 0 connected to the SST26 Driver
 - Instance 1 connected to the EFC peripheral library
- SST26 Driver
- QSPI peripheral library
- System Timer Service for Polling the status of at regular intervals
- TC peripheral library
- FreeRTOS

The components used may vary based on the project configuration selected:

- EFC peripheral library
- Board Support Package (BSP)

Other MHC Configurations:

- Enable Memory Protection Unit (MPU) for the QSPI Flash memory region
- Enable Memory Protection Unit (MPU) for reserving Internal Flash region for the NVM operations

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_e70_xult_freertos.X.

- **Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
- **Hardware Setup**
 - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED blinks when the data read from each media matches with the data written in them.

The following table provides the LED name:

Kit Name	LED Name
SAM E70 Xplained Ultra Evaluation Kit	LED 1

SDHC Driver Applications

Async

Name	Description
sdhc_read_write	This example uses SDHC driver to perform Block Erase/Write/Read operations on SD-Card attached to the device.

Description

This Section provides help for the SDHC Driver Applications.

Async

sdhc_read_write

This example uses SDHC driver to perform Block Erase/Write/Read operations on SD-Card attached to the device.

Description

This example uses the SDHC driver in asynchronous mode in both Bare-Metal and RTOS environment to perform Block Erase/Write/Read operations operations on SD-Card .

Building The Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/sdhc/async/sdhc_read_write/firmware

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE.

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_V71_xult.X	SAM V71 Xplained Ultra Evaluation Kit
sam_V71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This section provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components Used:

- SDHC Driver (Asynchronous mode)

- System Timer Service

The following components used may vary based on the project configuration selected:

- Board Support Package (BSP)
- FreeRTOS
-

Other MHC Settings:

- Configure the Pins for the HSMCI peripheral in the Pin Settings
- Configure XDMAC channels for transmit and receive

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project sam_e70_xult.X.**
2. **Project sam_e70_xult_freertos.X.**
 - **Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
 - **Hardware Setup**
 - Insert Micro-SD Card in the Micro-SD Card slot of the Device
3. **Project sam_V71_xult.X.**
4. **Project sam_V71_xult_freertos.X.**
 - **Hardware Used**
 - [SAM V71 Xplained Ultra Evaluation Kit](#)
 - **Hardware Setup**
 - Insert SD Card in the SD Card slot of the Device

Running The Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB to the DEBUG port.
2. Build and program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED is turned ON when the value is read from SDHC matches with the written data

The following table provides the LED name:

Kit Name	LED Name
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 0

SDSPI Driver Applications

Sync

Name	Description
sdspi_read_write	This application writes 10KB (10240 bytes) of data starting at the SD Card memory location 0x200, using the SDSPI driver APIs. The application then reads and verifies the written data.

Description

This section provides help for the SPI based SD Card (SDSPI) Driver applications.

Sync

[sdspi_read_write](#)

This application writes 10KB (10240 bytes) of data starting at the SD Card memory location 0x200, using the SDSPI driver APIs. The application then reads and verifies the written data.

Description

The SDSPI driver is configured to use the DMA to perform blocking read and write operations. A separate RTOS thread is created by the MHC to run the SDSPI task routine. This task routine checks for the SD Card attach/detach status and initializes the SD Card making it ready for the application to submit requests.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/sdspi/sync/sdspi_read_write/firmware
------------------	---

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE.

Project Name	Description
sam_c21n_xpro_freertos.X	SAM C21N Xplained Pro Evaluation Kit + FreeRTOS
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This section provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components Used:

- SDSPI Driver (Synchronous mode)

- DMA peripheral library
- Time System Service
- Timer peripheral library
- FreeRTOS

The following components used may vary based on the project configuration selected:

- SPI or SERCOM
- Board Support Package (BSP)

Other MHC Settings:

- Configure pins for the SPI Peripheral in Pin Settings
- Configure the SD Card Chip Select pin in Pin Settings

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project `sam_c21n_xpro_freertos.X`.

- **Hardware Used:**
 - SAM C21N Xplained Pro Evaluation Kit
 - IO1 Xplained PRO Extension Kit (for micro SD card)
- **Hardware Setup:**
 - Connect the [IO1 Xplained PRO Extension Kit](#) to the EXT1 header of the [SAM C21N Xplained Pro Evaluation Kit](#) as shown below:



- Insert micro SD card on the [IO1 Xplained PRO Extension Kit](#)

2. Project `sam_e70_xult_freertos.X`.

- **Hardware Used:**
 - SAM E70 Xplained Ultra Evaluation Kit
 - [IO1 Xplained PRO Extension Kit](#) (for micro SD card)
- **Hardware Setup:**
 - Connect the [IO1 Xplained PRO Extension Kit](#) to the EXT1 header of the SAM E70 Xplained Ultra Evaluation Kit as shown below:



- Insert micro SD card on the [IO1 Xplained PRO Extension Kit](#)

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB to the DEBUG port.
2. Build and program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED is turned ON when the value is read from SDSPI matches with the written data

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1

SPI Driver Applications

Async

Name	Description
spi_multi_instance	This example demonstrates how to use multiple instances of the SPI driver in asynchronous mode to communicate with multiple EEPROMs.
spi_multi_slave	This example demonstrates how to use single instance of the SPI driver in asynchronous mode to communicate with multiple EEPROMs.
spi_self_loopback_multi_client	This example demonstrates how to use the SPI driver in asynchronous mode to achieve self-loop back between multiple clients.

Sync

Name	Description
spi_multi_instance	This example demonstrates how to use multiple instances of the SPI driver in synchronous mode to communicate with multiple EEPROMs in RTOS environment.
spi_multi_slave	This example demonstrates how to use single instance of the SPI driver in synchronous mode to communicate with multiple EEPROMs in RTOS environment.
spi_self_loopback_multi_client	This example demonstrates how to use the SPI driver in synchronous mode to achieve self-loop back between multiple clients in RTOS environment.

Description

This section provides help for the SPI driver applications.

Async

spi_multi_instance

This example demonstrates how to use multiple instances of the SPI driver in asynchronous mode to communicate with multiple EEPROMs.

Description

This example write and read data to/from two separate EEPROM connected over two different SPI bus by using multi instance feature of the driver. It uses request (write and read request) queuing feature of asynchronous driver and do not waste CPU bandwidth in waiting for previous request completion. The example also demonstrates how to setup two different EEPROM transfers at different baud rates.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/spi/async/spi_multi_instance/firmware
------------------	--

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components used:

- SPI Driver (Asynchronous mode)

The following components used may vary based on the project configuration selected:

- SERCOM peripheral library
- Board Support Package (BSP)

Other MHC settings:

- Configure the SERCOM related Pins and SERCOM Chip Select Pins in the Pin Settings

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_c21n_xpro.X.

- **Hardware Used:**
 - SAM C21N Xplained Pro Evaluation Kit
 - Two EEPROM 4 Click boards
 - Two microBUS Xplained Pro boards
- **Hardware Setup:**
 - Install two EEPROM 4 Click boards on to the two microBUS Xplained Pro boards
 - Connect two microBUS Xplained Pro boards to the "EXT1" and "EXT2" headers of the SAM C21N Xplained Pro Evaluation Kit

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED blinks on Success and remains ON on Failure

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0

spi_multi_slave

This example demonstrates how to use single instance of the SPI driver in asynchronous mode to communicate with multiple EEPROMs.

Description

This example write and read data to/from two separate EEPROM connected over same the SPI bus by using multi client feature of the driver. It uses request (write and read request) queuing feature of asynchronous driver and do not waste CPU bandwidth in waiting for previous request completion. The example also demonstrates how to setup two different EEPROM transfers at different baud rates.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/spi/async/spi_multi_slave/firmware

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult.X	SAM V71 Xplained Ultra Evaluation Kit
sam_v71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components used:

- SPI Driver (Asynchronous mode)

The following components used may vary based on the project configuration selected:

- Board Support Package (BSP)
- FreeRTOS

- Other MHC settings:
 - Configure the Pins for the USART peripheral in the Pin Settings
 - Configure the SPI Chip Select, Hold and Write Protect Pins for both the EEPROMs
 - Enable DMA for Transmit and Receive (DMA Channels for transmit and receive automatically gets allocated)

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project **sam_e70_xult.X**.

2. Project **sam_e70_xult_freertos.X**.

- Hardware Used:**

- SAM E70 Xplained Ultra Evaluation Kit
- Two [EEPROM 4 Click](#) boards
- [microBUS Xplained Pro](#) board

- Hardware Setup:**

- Install one [EEPROM 4 Click](#) board on to the click slot provided on SAM E70 Xplained Ultra Evaluation Kit
- Install another [EEPROM 4 Click](#) board on to the [microBUS Xplained Pro](#) board
- Connect [microBUS Xplained Pro](#) board to the EXT2 headers of the SAM E70 Xplained Ultra Evaluation Kit

3. Project **sam_v71_xult.X**.

4. Project **sam_v71_xult_freertos.X**.

- Hardware Used:**

- [SAM V71 Xplained Ultra Evaluation Kit](#)
- Two [EEPROM 4 Click](#) boards
- Two [microBUS Xplained Pro](#) boards

- Hardware Setup:**

- Install [EEPROM 4 Click](#) boards on to the [microBUS Xplained Pro](#) boards

- Connect [microBUS Xplained Pro](#) boards on to the "EXT1" and "EXT2" headers of the [SAM V71 Xplained Ultra Evaluation Kit](#)

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED turns ON on Success and remains OFF on Failure

The following table provides the LED name:

Kit Name	LED Name
SAM E70 Xplained Ultra Evaluation Kit	LED 2
SAM V71 Xplained Ultra Evaluation Kit	LED 0

Note:

Make sure a jumper is connected between **LED2** and PB08 on J204 header of SAM E70 Xplained Ultra Evaluation Kit.

spi_self_loopback_multi_client

This example demonstrates how to use the SPI driver in asynchronous mode to achieve self-loop back between multiple clients.

Description

This example write and read back the same data (self loop back) for two different clients connected over same the SPI bus by using multi client feature of the driver. It uses request (write and read request) queuing feature of asynchronous driver and do not waste CPU bandwidth in waiting for previous request completion. The example also demonstrates how to setup two different client transfers at different baud rates.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/spi/async/spi_self_loopback_multi_client/firmware

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_d20_xpro.X	SAM D20 Xplained Pro Evaluation Kit
sam_d21_xpro.X	SAM D21 Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freetos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult.X	SAM V71 Xplained Ultra Evaluation Kit

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components used:

- SPI Driver (Asynchronous mode)

The following components used may vary based on the project configuration selected:

- SERCOM peripheral library in Interrupt mode
- Board Support Package (BSP)
- FreeRTOS

Other MHC settings:

- Configure the Pins for the SERCOM peripheral in the Pin Settings
- Configure the SPI Chip Select Pins in the Pin Settings

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_c21n_xpro.X.

- **Hardware Used:**
 - [SAM C21N Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Use jumper wire to connect "Pin 16 of EXT2 header" to "Pin 17 of EXT2 header"
 - SERCOM5 PAD2 signal as MOSI signal and it is mapped to PB00 that is routed to "Pin 16 of EXT2 header"
 - SERCOM5 PAD0 signal as MISO signal and it is mapped to PB02 that is routed to "Pin 17 of EXT2 header"

2. Project sam_d20_xpro.X.

3. Project sam_d21_xpro.X.

- **Hardware Used:**
 - [SAM D20 Xplained Pro Evaluation Kit](#) or [SAM D21 Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Use jumper wire to Connect "Pin 16 of EXT2 header" to "Pin 17 of EXT2 header"
 - SERCOM1 PAD2 signal as MOSI signal and it is mapped to PA18 that is routed to "Pin 16 of EXT2 header"
 - SERCOM1 PAD0 signal as MISO signal and it is mapped to PA16 that is routed to "Pin 17 of EXT2 header"

4. Project sam_e70_xult.X.

5. Project sam_e70_xult_freertos.X.

- **Hardware Used:**
 - SAM E70 Xplained Ultra Evaluation Kit
- **Hardware Setup:**
 - Use jumper wire to Connect "Pin 16 of EXT1 header" to "Pin 17 of EXT1 header"

- SPI0 MOSI signal is mapped to PD21 that is routed to "Pin 16 of EXT1 header"
- SPI0 MISO signal is mapped to PD20 that is routed to "Pin 17 of EXT1 header"

6. Project sam_v71_xult.X.

7. Project sam_v71_xult_freertos.X.

- **Hardware Used:**

- [SAM V71 Xplained Ultra Evaluation Kit](#)

- **Hardware Setup:**

- Use jumper wire to Connect "Pin 16 of EXT1 header" to "Pin 17 of EXT1 header"
- SPI0 MOSI signal is mapped to PD21 that is routed to "Pin 16 of EXT1 header"
- SPI0 MISO signal is mapped to PD20 that is routed to "Pin 17 of EXT1 header"

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED turns ON on Success and remains OFF on Failure

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM D20 Xplained Pro Evaluation Kit	LED 0
SAM D21 Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 0

Sync

spi_multi_instance

This example demonstrates how to use multiple instances of the SPI driver in synchronous mode to communicate with multiple EEPROMs in RTOS environment.

Description

This example write and read data to/from two separate the EEPROM connected over different the SPI buses by using multi instance feature of synchronous SPI driver. The example also demonstrates how to setup two different instance transfers at different baud rates.

The example has three RTOS threads for the purpose:

1. **APP_EEPROM1_Tasks:** This thread starts the EEPROM write-read operation , once the transfer is successfully completed. It repeats the same step continuously.
2. **APP_EEPROM2_Tasks:** This thread starts the EEPROM write-read operation , once the transfer is successfully completed. It repeats the same step continuously.
3. **APP_MONITOR_Tasks:** This thread checks the status of transfers done by first two threads and toggle LED. It then blocks for 100ms so that other threads can run. It repeats the same after 100ms.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/spi/sync/spi_multi_instance/firmware
------------------	---

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro_freertos.X	SAM C21N Xplained Pro Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components used:

- SPI Driver (Synchronous mode)

The following components used may vary based on the project configuration selected:

- SPI peripheral library in Interrupt mode
- Board Support Package (BSP)
- FreeRTOS

Other MHC settings:

- Configure the Pins for the SPI peripheral in the Pin Settings
- Configure the SPI Chip Select, Hold and Write Protect Pins of both the EEPROMs
- Enable DMA for Transmit and Receive (DMA Channels for transmit and receive automatically gets allocated)

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_c21n_xpro.X.

• Hardware Used:

- SAM C21N Xplained Pro Evaluation Kit
- Two EEPROM 4 Click boards
- Two microBUS Xplained Pro boards

• Hardware Setup:

- Install two EEPROM 4 Click boards on to the two microBUS Xplained Pro boards
- Connect two microBUS Xplained Pro boards to the "EXT1" and "EXT2" headers of the SAM C21N Xplained Pro Evaluation Kit

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED blinks on Success

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0

spi_multi_slave

This example demonstrates how to use single instance of the SPI driver in synchronous mode to communicate with multiple EEPROMs in RTOS environment.

Description

This example write and read data to/from two separate the EEPROM connected over same the SPI bus by using multi client feature of synchronous SPI driver. The example also demonstrates how to setup two different client transfers at two different baud rates.

The example has three RTOS threads for the purpose:

1. **APP_EEPROM1_Tasks:** This thread starts the EEPROM write-read operation , once the transfer is successfully completed. It repeats the same step continuously.
2. **APP_EEPROM2_Tasks:** This thread starts the EEPROM write-read operation , once the transfer is successfully completed. It repeats the same step continuously.
3. **APP_MONITOR_Tasks:** This thread checks the status of transfers done by first two threads and toggle LED. It then blocks for 1000ms so that other threads can run. It repeats the same after 1000ms.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/spi/sync/spi_multi_slave/firmware

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components used:

- SPI Driver (Synchronous mode)

The following components used may vary based on the project configuration selected:

- SPI peripheral library in Interrupt mode
- Board Support Package (BSP)
- FreeRTOS

Other MHC settings:

- Configure the Pins for the SPI peripheral in the Pin Settings
- Configure the SPI Chip Select, Hold and Write Protect Pins of both the EEPROMs
- Enable DMA for Transmit and Receive (DMA Channels for transmit and receive automatically gets allocated)

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_e70_xult_freertos.X.

• Hardware Used:

- SAM E70 Xplained Ultra Evaluation Kit
- Two [EEPROM 4 Click](#) boards
- [microBUS Xplained Pro](#) board

• Hardware Setup:

- Install one [EEPROM 4 Click](#) board on to the click slot provided on SAM E70 Xplained Ultra Evaluation Kit
- Install another [EEPROM 4 Click](#) board on to the [microBUS Xplained Pro](#) board
- Connect [microBUS Xplained Pro](#) board to the EXT2 headers of the SAM E70 Xplained Ultra Evaluation Kit

2. Project sam_v71_xult_freertos.X.

• Hardware Used:

- [SAM E70 Xplained Ultra Evaluation Kit](#)
- Two [EEPROM 4 Click](#) boards
- [microBUS Xplained Pro](#) board

• Hardware Setup:

- Install [EEPROM 4 Click](#) boards on to the [microBUS Xplained Pro](#) boards
- Connect [microBUS Xplained Pro](#) boards on to the "EXT1" and "EXT2" headers of the [SAM V71 Xplained Ultra Evaluation Kit](#)

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED blinks on Success

The following table provides the LED name:

Kit Name	LED Name
SAM E70 Xplained Ultra Evaluation Kit	LED 2
SAM V71 Xplained Ultra Evaluation Kit	LED 0

Note:

Make sure a jumper is connected between **LED2** and PB08 on J204 header of SAM E70 Xplained Ultra Evaluation Kit.

spi_self_loopback_multi_client

This example demonstrates how to use the SPI driver in synchronous mode to achieve self-loop back between multiple clients in RTOS environment.

Description

This example write and read back the same data (self loop back) for two different clients connected over same SPI bus by using multi client feature of synchronous SPI driver. The example also demonstrates how to setup two different client transfers at two different baud rates.

The example has three RTOS threads for the purpose:

- APP_CLIENT1_Tasks:** This thread starts first loop back transfer, once the transfer is successfully completed, it blocks for 100ms so that other threads can run. It repeats the same after 100ms.
- APP_CLIENT2_Tasks:** This thread waits for its chance, starts second loop back transfer, once the transfer is successfully completed, it blocks for 100ms so that other threads can run. It repeats the same after 100ms.
- APP_MONITOR_Tasks:** This thread checks the status of transfers done by first two threads and toggle LED. It then blocks for 100ms so that other threads can run. It repeats the same after 100ms.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/spi/sync/spi_self_loopback_multi_client/firmware
------------------	---

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro_freertos.X	SAM C21N Xplained Pro Evaluation Kit + FreeRTOS
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components used:

- SPI Driver (Synchronous mode)

The following components used may vary based on the project configuration selected:

- SPI peripheral library in Interrupt mode
- SERCOM peripheral library in Interrupt mode
- Board Support Package (BSP)
- FreeRTOS

Other MHC settings:

- Configure the Pins for the SPI peripheral in the Pin Settings
- Configure the SPI Chip Select, Hold and Write Protect Pins of both the EEPROMs
- Enable DMA for Transmit and Receive (DMA Channels for transmit and receive automatically gets allocated)

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_c21n_xpro.X.

- **Hardware Used:**
 - [SAM C21N Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Use jumper wire to Connect "Pin 16 of EXT2 header" to "Pin 17 of EXT2 header"
 - SERCOM5 PAD2 signal as MOSI signal and it is mapped to PB00 that is routed to "Pin 16 of EXT2 header"
 - SERCOM5 PAD0 signal as MISO signal and it is mapped to PB02 that is routed to "Pin 17 of EXT2 header"

2. Project sam_e70_xult_freertos.X.

- **Hardware Used:**
 - SAM E70 Xplained Ultra Evaluation Kit
- **Hardware Setup:**
 - Use jumper wire to Connect "Pin 16 of EXT1 header" to "Pin 17 of EXT1 header"
 - SPI0 MOSI signal is mapped to PD21 that is routed to "Pin 16 of EXT1 header"
 - SPI0 MISO signal is mapped to PD20 that is routed to "Pin 17 of EXT1 header"

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED blinks on Success

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1

SST26 Driver Applications

This section provides help for the SST26 driver applications.

sst26_flash_read_write

This example application shows how to use the SST26 driver to perform block operations on the On-Board QSPI FLASH memory.

Description

This application uses the SST26 driver to Erase/Write/Read on the On-Board QSPI FLASH memory using the QSPI peripheral library.

The application consists of APP_SST26_Tasks() which is called through SYS_Tasks() routine.

Building The Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/driver/sst26/sst26_flash_read_write/firmware
------------------	--

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

The components used may vary based on the project configuration selected:

- SST26 Driver
- QSPI peripheral library
- SYSTICK peripheral library

The following components used may vary based on the project configuration selected:

- Board Support Package (BSP)

Other MHC Configurations:

- Enable Memory Protection Unit (MPU) for the QSPI Flash memory region

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_e70_xult.X.

- **Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
- **Hardware Setup**
 - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED blinks when the data read from each media matches with the data written in them.

The following table provides the LED name:

Kit Name	LED Name
SAM E70 Xplained Ultra Evaluation Kit	LED 1

USART Driver Applications

Async

Name	Description
uart_echo	This example echoes the received characters over the console using the USART driver in asynchronous mode.
uart_multi_instance	This example echoes the received characters over the two consoles using USART driver in asynchronous mode.

Sync

Name	Description
uart_echo	This example echoes the received characters over the console using the USART driver in synchronous mode.
uart_multi_instance	This example echoes the received characters over the two consoles using USART driver in synchronous mode with DMA enabled.

Description

This section provides help for the USART Driver applications.

Async

uart_echo

This example echoes the received characters over the console using the USART driver in asynchronous mode.

Description

This example uses the USART driver in asynchronous mode in both Bare-Metal and RTOS environment to communicate over the console. It receives and echo's back the characters entered by the user.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core\apps\driver\uart\async\uart_echo\firmware
------------------	--

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE.

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult.X	SAM V71 Xplained Ultra Evaluation Kit
sam_v71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configurations

This section provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components Used:

- USART Driver (Asynchronous mode)

The following components used may vary based on the project configuration selected:

- USART or SERCOM peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Settings:

- Configure the Pins for the USART or SERCOM peripheral in the Pin Settings

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_c21n_xpro.X.

- **Hardware Used:**
 - SAM C21N Xplained Pro Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup required

2. Project sam_e70_xult.X.

3. Project sam_e70_xult_freertos.X.

- **Hardware Used:**
 - SAM E70 Xplained Ultra Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup required

4. Project sam_v71_xult.X.

5. Project sam_v71_xult_freertos.X.

- **Hardware Used:**
 - SAM V71 Xplained Ultra Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup required

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB to the DEBUG port.

2. Open Terminal application (Ex.:Tera Term) on the computer.

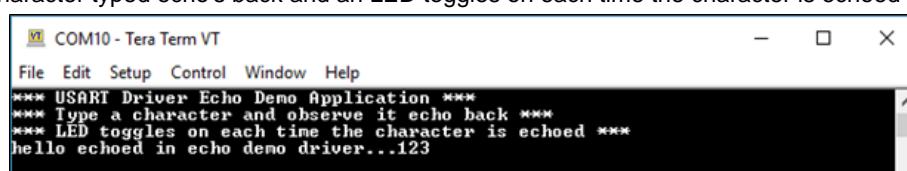
3. Configure the serial port settings as follows:

- Baud : 115200
- Data : 8 Bits
- Parity : None
- Stop : 1 Bit
- Flow Control : None

4. Build and program the application using the MPLAB X IDE.

5. Type a character and observe the output on the console as shown below:

- If success the character typed echo's back and an LED toggles on each time the character is echoed



The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 1

uart_multi_instance

This example echoes the received characters over the two consoles using USART driver in asynchronous mode.

Description

This example uses the USART driver in asynchronous mode in both Bare-Metal and RTOS environment to communicate over two consoles. It receives and echo's back the characters entered by the user on the respective console.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core\apps\driver\uart\async\uart_multi_instance\firmware
------------------	--

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE.

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_d20_xpro.X	SAM D20 Xplained Pro Evaluation Kit
sam_d21_xpro.X	SAM D21 Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult.X	SAM V71 Xplained Ultra Evaluation Kit
sam_v71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configurations

This section provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components Used:

- USART Driver (Asynchronous mode)

The following components used may vary based on the project configuration selected:

- Two USART or SERCOM peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Settings:

- Configure the Pins for the USART or SERCOM peripherals in the Pin Settings

Hardware Setup

This section describes how to configure the supported hardware.

Description

Common Hardware Setup:

- Install an [USB UART click board](#) on to the [microBUS Xplained Pro](#) board

1. Project [sam_c21n_xpro.X](#).

- **Hardware Used:**
 - [SAM C21N Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT1 header of the [SAM C21N Xplained Pro Evaluation Kit](#)
 - Connect mini USB to the [microBUS Xplained Pro](#) board

2. Project [sam_d20_xpro.X](#).

- **Hardware Used:**
 - [SAM D20 Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT1 header of the [SAM D20 Xplained Pro Evaluation Kit](#)
 - Connect mini USB to the [microBUS Xplained Pro](#) board

3. Project [sam_d21_xpro.X](#).

- **Hardware Used:**
 - [SAM D21 Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT1 header of the [SAM D21 Xplained Pro Evaluation Kit](#)
 - Connect mini USB to the [microBUS Xplained Pro](#) board

4. Project [sam_e70_xult.X](#).

5. Project [sam_e70_xult_freertos.X](#).

- **Hardware Used:**
 - [SAM E70 Xplained Ultra Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT1 header of the [SAM E70 Xplained Ultra](#) board
 - Connect mini USB to the [microBUS Xplained Pro](#) board

5. Project [sam_v71_xult.X](#).

6. Project [sam_v71_xult_freertos.X](#).

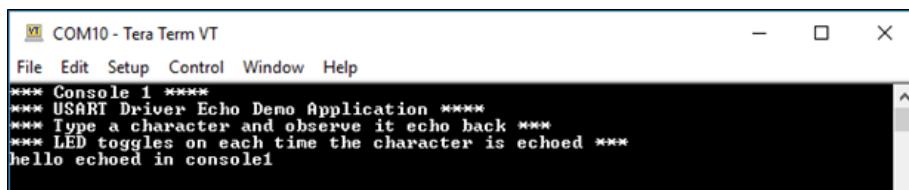
- **Hardware Used:**
 - [SAM V71 Xplained Ultra Evaluation Kit](#)
- **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT1 header of the [SAM V71 Xplained Ultra](#) board
 - Connect mini USB to the [microBUS Xplained Pro](#) board

Running the Application

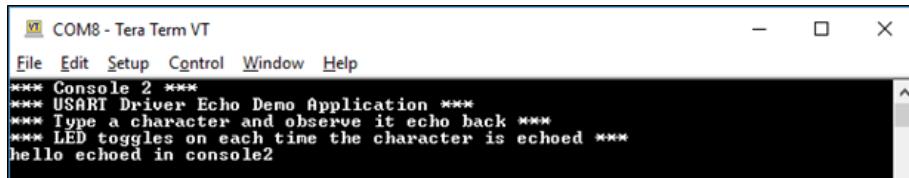
This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB to the DEBUG port. This will enumerate a first port.
2. Connect a mini USB to the [microBUS Xplained Pro](#) board. This will enumerate a second port.
3. Open Terminal application (Ex.:Tera Term) on the computer for two COM ports.
4. Configure the serial port settings as follows:
 - Baud : 115200
 - Data : 8 Bits
 - Parity : None
 - Stop : 1 Bit
 - Flow Control : None
5. Build and program the application using the MPLAB X IDE.
6. Type a character and observe the output on the two consoles as shown below:
 - If success the character typed echo's back and an LED toggles on each time the character is echoed
 - Console 1:



- Console 2:



The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM D20 Xplained Pro Evaluation Kit	LED 0
SAM D21 Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 1

Sync

uart_echo

This example echoes the received characters over the console using the USART driver in synchronous mode.

Description

This example uses the USART driver in synchronous mode in RTOS environment to communicate over the console. It receives and echo's back the characters entered by the user.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core\apps\driver\uart\sync\uart_echo\firmware
-------------------------	--

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE.

Project Name	Description
sam_c21n_xpro_freertos.X	SAM C21N Xplained Pro Evaluation Kit + FreeRTOS
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configurations

This section provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components Used:

- USART Driver (Synchronous mode)
- FreeRTOS

The following components used may vary based on the project configuration selected:

- USART or SERCOM peripheral library
- Board Support Package (BSP)

Other MHC Settings:

- Configure the Pins for the USART or SERCOM peripheral in the Pin Settings

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_c21n_xpro_freertos.X.

- **Hardware Used:**
 - SAM C21N Xplained Pro Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup required

2. Project sam_e70_xult_freertos.X.

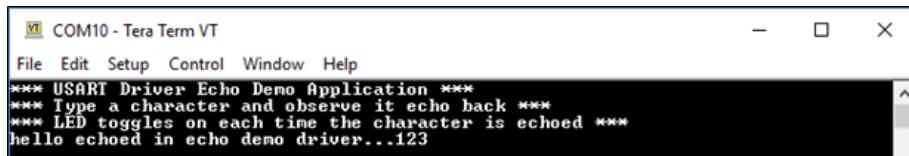
- **Hardware Used:**
 - SAM E70 Xplained Ultra Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup required

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB to the DEBUG port.
2. Open Terminal application (Ex.:Tera term) on the computer.
3. Configure the serial port settings as follows:
 - Baud : 115200
 - Data : 8 Bits
 - Parity : None
 - Stop : 1 Bit
 - Flow Control : None
4. Build and program the application using the MPLAB X IDE.
5. Type a character and observe the output on the console as shown below:
 - If success the character typed echo's back and an LED toggles on each time the character is echoed



The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1

uart_multi_instance

This example echoes the received characters over the two consoles using USART driver in synchronous mode with DMA enabled.

Description

This example uses the USART driver in synchronous mode in RTOS environment with DMA enabled to communicate over two consoles. It receives and echo's back the characters entered by the user on the respective console.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core\apps\driver\uart\sync\uart_multi_instance\firmware
------------------	---

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE.

Project Name	Description
sam_c21n_xpro_freertos.X	SAM C21N Xplained Pro Evaluation Kit + FreeRTOS
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configurations

This section provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components Used:

- USART Driver (Synchronous mode)
- FreeRTOS

The following components used may vary based on the project configuration selected:

- Two USART or SERCOM peripheral library
- Board Support Package (BSP)

Other MHC Settings:

- Configure the Pins for the USART or SERCOM peripherals in the Pin Settings
- Configure the DMA transmit or receive channels for USART or SERCOM peripherals

Hardware Setup

This section describes how to configure the supported hardware.

Description

Common Hardware Setup:

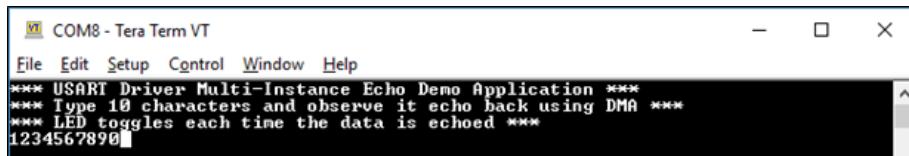
- Install an [USB UART click board](#) on to the [microBUS Xplained Pro](#) board
- 1. Project sam_c21n_xpro.X.**
- **Hardware Used:**
 - [SAM C21N Xplained Pro Evaluation Kit](#)
 - **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT1 header of the [SAM C21N Xplained Pro Evaluation Kit](#)
 - Connect mini USB to the [microBUS Xplained Pro](#) board
- 2. Project sam_e70_xult_freertos.X.**
- **Hardware Used:**
 - [SAM E70 Xplained Ultra Evaluation Kit](#)
 - **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT1 header of the [SAM C21N Xplained Pro Evaluation Kit](#)
 - Connect mini USB to the [microBUS Xplained Pro](#) board
- 3. Project sam_v71_xult_freertos.X.**
- **Hardware Used:**
 - [SAM V71 Xplained Ultra Evaluation Kit](#)
 - **Hardware Setup:**
 - Connect the [microBUS Xplained Pro](#) board to the EXT1 header of the [SAM C21N Xplained Pro Evaluation Kit](#)
 - Connect mini USB to the [microBUS Xplained Pro](#) board

Running the Application

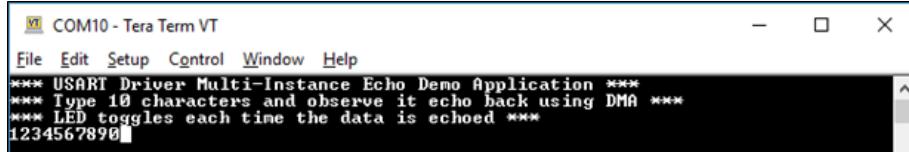
This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB to the DEBUG port. This will enumerate a first port.
2. Connect a mini USB to the [microBUS Xplained Pro](#) board. This will enumerate a second port.
3. Open Terminal application (Ex.:Tera Term) on the computer for two COM ports.
4. Configure the serial port settings as follows:
 - Baud : 115200
 - Data : 8 Bits
 - Parity : None
 - Stop : 1 Bit
 - Flow Control : None
5. Build and program the application using the MPLAB X IDE.
6. Type 10 characters and observe it echo back using DMA in two consoles as shown below:
 - If success the 10 characters typed echo's back and an LED toggles on each time the 10 characters are echoed
 - Console 1:



- Console 2:



The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 1

File System Applications

This section provides help for the File System applications.

nvm_fat

This application shows an example of implementing a FAT disk in the device internal Flash memory.

Description

File System Operations on NVM:

- The application contains a FAT disk image consisting of a Master Boot Record (MBR) sector, Logical Boot Sector, File Allocation Table, and Root Directory Area, placed in the internal Flash memory (NVM)
- The application opens an existing file named **FILE.TXT** and performs following file system related operations:
 - [SYS_FS_FileStat](#)
 - [SYS_FS_FileSize](#)

- [SYS_FS_FileSeek](#)
- [SYS_FS_FileEOF](#)
- Finally, the string "**Hello World**" is written to this file. The string is then read and compared with the string that was written to the file. If the string compare is successful, An LED indication is provided.

File system layer uses:

- Memory driver to communicate with underlying NVM media.

Building The Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/fs/nvm_fat/firmware
------------------	-------------------------------

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_d20_xpro.X	SAM D20 Xplained Pro Evaluation Kit
sam_d21_xpro.X	SAM D21 Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_V71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- File System configured to use FAT Format
- Memory Driver connected to EFC or NVMCTRL peripheral library
 - For Bare-Metal it is configured to asynchronous mode
 - For RTOS it is configured to synchronous mode

The following components used may vary based on the project configuration selected:

- EFC or NVMCTRL peripheral library
- Timer System Service only when Memory driver is configured in synchronous mode
- TC peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Configurations:

- Enable MPU for reserving Internal Flash region for NVM media operations if cache is enabled

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_c21n_xpro.X.

- **Hardware Used**
 - SAM C21N Xplained Pro Evaluation Kit
- **Hardware Setup**
 - No Special hardware Setup Required.

2. Project sam_d20_xpro.X.

- **Hardware Used**
 - SAM D20 Xplained Pro Evaluation Kit
- **Hardware Setup**
 - No Special hardware Setup Required.

3. Project sam_d21_xpro.X.

- **Hardware Used**
 - SAM D21 Xplained Pro Evaluation Kit
- **Hardware Setup**
 - No Special hardware Setup Required.

4. Project sam_e70_xult.X.

5. Project sam_e70_xult_freertos.X.

- **Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
- **Hardware Setup**
 - No Special hardware Setup Required.

6. Project sam_v71_xult_freertos.X.

- **Hardware Used**
 - SAM V71 Xplained Ultra Evaluation Kit
- **Hardware Setup**
 - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using the MPLAB X IDE.
3. The LED is turned ON when the File "FILE.TXT" has the app data "**Hello World**" written in it

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM D20 Xplained Pro Evaluation Kit	LED 0

SAM D21 Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 0

nvm_mpfs

This application shows an example of implementing a MPFS disk in device Internal Flash memory.

Description

File System Operations on NVM:

- The application contains a MPFS disk image in the internal Flash memory. The disk image contains two files named:
 - FILE.txt**, Size = **11 Bytes**. The content of the file is: "**Hello World**"
 - TEST.txt**, Size = **10 Bytes**. The content of the file is: "**1234567890**"
- The application performs following file system related operations:
 - SYS_FS_FileRead**
 - SYS_FS_FileStat**
 - SYS_FS_FileSize**
 - SYS_FS_FileSeek**
 - SYS_FS_FileEOF**
- The contents of both the files are read and compared with the expected strings as mentioned above. If the string compare is successful, An LED indication is provided.

File system layer uses:

- Memory driver to communicate with underlying NVM media.

Building The Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/fs/nvm_mpfs/firmware
------------------	---------------------------------------

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_V71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- File System configured to use MPFS Format
- Memory Driver connected to EFC or NVMCTRL peripheral library
 - For Bare-Metal it is configured to asynchronous mode
 - For RTOS it is configured to synchronous mode

The following components used may vary based on the project configuration selected:

- EFC or NVMCTRL peripheral library
- Timer System Service only when Memory driver is configured in synchronous mode
- TC peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Configurations:

- Enable MPU for reserving Internal Flash region for NVM media operations if cache is enabled

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_c21n_xpro.X.

- **Hardware Used**
 - SAM C21N Xplained Pro Evaluation Kit
- **Hardware Setup**
 - No Special hardware Setup Required.

2. Project sam_e70_xult.X.

3. Project sam_e70_xult_freertos.X.

- **Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
- **Hardware Setup**
 - No Special hardware Setup Required.

4. Project sam_v71_xult_freertos.X.

- **Hardware Used**
 - SAM V71 Xplained Ultra Evaluation Kit
- **Hardware Setup**
 - No Special hardware Setup Required.

Running The Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using the MPLAB X IDE.
3. The LED is turned ON when the following criteria is satisfied.
 - File "FILE.txt" has the string "**Hello World**" in it
 - File "TEST.txt" has the string "1234567890" in it

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 0

nvm_sdhc_fat_multi_disk

This application shows an example of using the MPLAB Harmony File System to access files across multiple media (NVM, SDHC).

Description

File System Operations on NVM and SD-Card:

- The application contains a FAT disk image consisting of a Master Boot Record (MBR) sector, Logical Boot Sector, File Allocation Table, and Root Directory Area, placed in the internal Flash memory (NVM)
- A SD card is used as another disk, which might have FAT16 or FAT32 implemented on it (dependent on the formatting of SD card)
- The application searches the NVM media for a file named **FILE.TXT**, opens and reads the contents of the file in NVM and copies the contents to the file, **FILE.TXT**, in the SD card
- Once the copy is successful, an addition string "**Test is successful**" is added to the file. If the write operation is successful, LED indication is provided

File system layer uses:

- Memory driver to communicate with underlying NVM media.
- SDHC Driver to communicate to SD-Card

Building The Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/fs/nvm_sdhc_fat_multi_disk/firmware

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_V71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- File System:
 - FAT Format for SDCARD
 - FAT Format for NVM
- Memory Driver connected to NVM Media
 - For Bare-Metal it is configured to asynchronous Mode
 - For RTOS it is configured to synchronous Mode
- SDHC Driver
- Timer System Service
- TC peripheral library

The following components used may vary based on the project configuration selected:

- EFC or NVMCTRL peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Configurations:

- Enable MPU for reserving Internal Flash region for NVM media operations
- Enable XDMAC channels for SDHC Transmit/Receive

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. **Project sam_e70_xult.X.**
2. **Project sam_e70_xult_freertos.X.**
 - **Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
 - **Hardware Setup**
 - Create a file named **FILE.txt** in a micro SD-Card containing some random data.
 - Insert the micro-SD Card in the micro-SD Card slot of the Device.
3. **Project sam_V71_xult_freertos.X.**
 - **Hardware Used**
 - **SAM V71 Xplained Ultra Evaluation Kit**
 - **Hardware Setup**
 - Create a file named **FILE.txt** in SD-Card containing some random data.
 - Insert the SD-Card in the SD-Card slot of the Device.

Running The Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using the MPLAB X IDE.
3. Insert the SD-Card in SD card Slot of the kit.
4. The LED is turned ON when the content from the NVM media is copied to SD-Card successfully.

The following table provides the LED name:

Kit Name	LED Name
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 0

5. If Success, Insert the SD-Card on to your host PC
6. **FILE.txt** should have the content "**This data from NVM Disk Test is successful**".

nvm_sdspi_fat_multi_disk

This application shows an example of using the MPLAB Harmony File System to access files across multiple media (NVM, SDSPI).

Description

File System Operations on NVM and SD-Card:

- The application contains a FAT disk image consisting of a Master Boot Record (MBR) sector, Logical Boot Sector, File Allocation Table, and Root Directory Area, placed in the internal Flash memory (NVM)
- A SD card is used as another disk, which might have FAT16 or FAT32 implemented on it (dependent on the formatting of SD card)
- The application searches the NVM media for a named **FILE.TXT**, opens and reads the contents of the file in NVM and copies the contents to the file, **FILE.TXT**, in the SD card
- Once the copy is successful, an addition string "**Test is successful**" is added to the file. If the write operation is successful, LED indication is provided

File system layer uses:

- Memory driver to communicate with underlying NVM media.
- SDSPI Driver to communicate to SD-Card over SPI.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/fs/nvm_sdspi_fat_multi_disk.sync/firmware

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro_freertos.X	SAM C21N Xplained Pro Evaluation Kit + FreeRTOS
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

The components used may vary based on the project configuration selected:

- File System
 - FAT Format for SDCARD
 - FAT Format for NVM
- SD SPI driver configured to synchronous mode.
- Memory driver connected to NVM Media and configured to synchronous mode.
- Timer System Service
- TC peripheral library

The following components used may vary based on the project configuration selected:

- EFC or NVMCTRL peripheral library
- SPI or SERCOM peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Configurations:

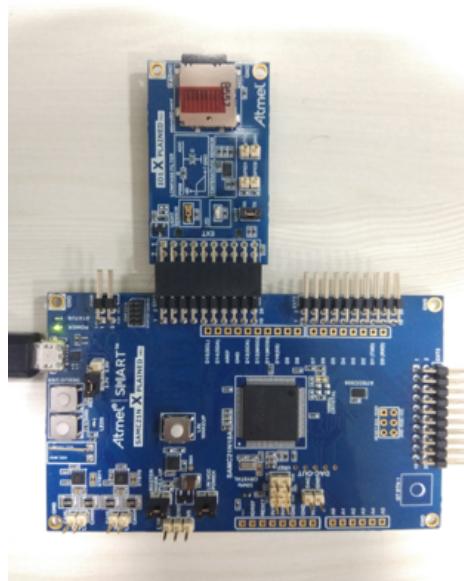
1. Enable MPU for reserving Internal Flash region for NVM Media operations if cache is enabled.
2. Configure the SD Card chip select pin in Pin Settings

Hardware Setup

This section describes how to configure the supported hardware.

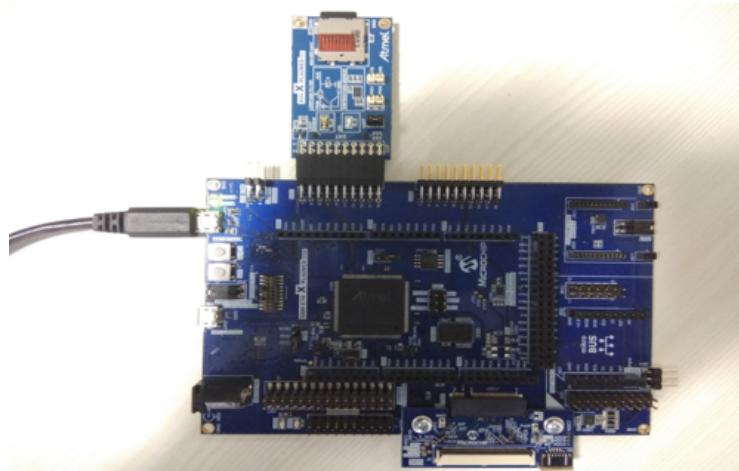
Description

1. Project [sam_c21n_xpro_freertos.X](#).
 - **Hardware Used**
 - [SAM C21N Xplained Pro Evaluation Kit](#)
 - [IO1 Xplained Pro Extension Kit](#)
 - **Hardware Setup**
 - Connect [IO1 Xplained Pro Extension Kit](#) to the EXT1 header of [SAM C21N Xplained Pro Evaluation Kit](#)
 - Create a file named **FILE.txt** in a micro SD-Card containing some random data.
 - Insert the micro SD-Card in the micro SD-Card slot of the IO1 Xplained Pro Extension Kit



2. Project sam_e70_xult_freertos.X.

- **Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
 - [IO1 Xplained Pro Extension Kit](#)
- **Hardware Setup**
 - Connect [IO1 Xplained Pro Extension Kit](#) to the EXT1 connector of SAM E70 Xplained Pro Evaluation Kit.
 - Create a file named **FILE.txt** in a micro SD-Card containing some random data.
 - Insert the micro SD-Card in the micro SD-Card slot of the [IO1 Xplained Pro Extension Kit](#)



Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using the MPLAB X IDE.
3. Insert the SD-Card in SD card Slot of the kit.
4. The LED is turned ON when the content from the NVM media is copied to SD-Card successfully.

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1

5. If Success, Insert the SD-Card on to your host PC
6. **FILE.txt** should have the content "**This data from NVM Disk Test is successful**".

nvm_sst26_fat

This application shows an example of using the MPLAB Harmony File System to access multiple files across multiple media (NVM, SST26 FLASH).

Description

File System Operations on NVM:

- The application contains a FAT disk image consisting of a Master Boot Record (MBR) sector, Logical Boot Sector, File Allocation Table, and Root Directory Area, placed in the internal Flash memory (NVM)

- The application opens an existing file named **FILE.TXT** and performs following file system related operations:
 - **SYS_FS_FileStat**
 - **SYS_FS_FileSize**
 - **SYS_FS_FileSeek**
 - **SYS_FS_FileEOF**
- Finally, the string "**Hello World**" is written to this file. The string is then read and compared with the string that was written to the file

File System Operations on the On-Board SST26 Flash Memory:

- Performs a SYS-FS_FormatDisk.
- Opens a **newfile.txt** on the sst26 flash
- Write and reads back 4KB of data on **newfile.txt**
- Verifies the Data Read back

File system layer uses:

- One instance of the Memory driver is used to communicate with underlying NVM media
- One instance of the Memory driver is used to communicate with the On-Board SST26 Flash memory

The Application Consists of 3 tasks which are called in RTOS Thread Context:

1. APP_SST26_Tasks() : Performs File Operations on SST26 QSPI Flash memory.
2. APP_NVM_Tasks() : Performs File Operations on NVM
3. APP_MONITOR_Tasks(): Monitors the state of above two Tasks.

Building The Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/fs/nvm_sst26_fat/firmware
------------------	-------------------------------------

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

The components used may vary based on the project configuration selected:

- File System:
 - FAT Format for SST26 Flash
 - FAT Format for NVM
- Memory Driver 2 Instances in synchronous Mode

- Instance 0 connected to SST26 Driver
- Instance 1 connected to EFC PLIB
- SST26 Driver
- QSPI peripheral library

The following components used may vary based on the project configuration selected:

- EFC or NVMCTRL peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Configurations:

- Enable MPU for reserving Internal Flash region for NVM media operations
- Enable MPU for QSPI Flash region for SST26 media operations

Hardware Setup

This section describes how to configure the supported hardware.

Description

- Project **sam_e70_xult_freertos.X**.
 - **Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
 - **Hardware Setup**
 - No Special hardware Setup Required.

Running The Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using the MPLAB X IDE.
3. The LED toggles when the following criteria is satisfied.
 - For NVM media: File "**FILE.TXT**" has the app data "**Hello World**" written in it
 - For SST26 media: 4KB of data has successfully been written and read back from file "**newfile.txt**"

The following table provides the LED name:

Kit Name	LED Name
SAM E70 Xplained Ultra Evaluation Kit	LED 1

qspi_sst26_fat

This application shows an example of using the MPLAB Harmony File System to access SST26 media.

Description

File System Operations on the On-Board SST26 Flash Memory:

- Performs a SYS-FS_FormatDisk
- Opens a **newfile.txt** on the sst26 flash
- Write and reads back 4KB of data on **newfile.txt**
- Verifies the Data Read back

File system layer uses:

- Memory driver to communicate with the On-Board SST26 QSPI Flash Memory

Building The Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	<code>core/apps/fs/qspi_sst26_fat/firmware</code>
------------------	---

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

The components used may vary based on the project configuration selected:

- File System:
 - FAT Format for SST26 Flash
- Memory Driver connected to SST26 Driver
 - For Bare-Metal it is configured to asynchronous mode
 - For RTOS it is configured to synchronous mode
- SST26 Driver
- QSPI peripheral library

The following components used may vary based on the project configuration selected:

- System Timer Service for Polling the status of at regular intervals for RTOS configuration
- TC peripheral library
- Board Support Package (BSP)
- FreeRTOS

Other MHC Configurations:

- Enable MPU for QSPI Flash region for SST26 Media operations

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project **sam_e70_xult.X**.
2. Project **sam_e70_xult_freertos.X**.
 - **Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
 - **Hardware Setup**
 - No Special hardware Setup Required.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using MPLAB X IDE.
3. The LED is turned ON if data has successfully been written and read back from file "**newfile.txt**"

The following table provides the LED name:

Kit Name	LED Name
SAM E70 Xplained Ultra Evaluation Kit	LED 1

sdhc_fat

This application shows an example of using the MPLAB Harmony File System to access and modify the contents of a SD card using the SDHC driver.

Description

File System Operations on the SD-Card:

- The application opens a file named **FILE_TOO_LONG_NAME_EXAMPLE_123.JPG** on the SD card
- Reads the content of the file and creates a directory named **Dir1**
- Inside the directory, writes the copied content into another file **FILE_TOO_LONG_NAME_EXAMPLE_123_1.JPG**

The image file could be any arbitrary JPEG (image) file chosen by the user and then renamed to **FILE_TOO_LONG_NAME_EXAMPLE_123.JPG**.

The reason for choosing a JPEG file for test purposes is that the duplicate file, **FILE_TOO_LONG_NAME_EXAMPLE_123_1.JPG** created by the demonstration could be easily verified for correctness.

If the new file inside **Dir1** opens for viewing on the computer and matches to original image, the test is deemed to have passed. Otherwise, if the file does not open (i.e., is corrupted), the test will be considered to have failed.

Note:

Since the application creates a directory named **Dir1**, it is important that the a folder with the same name does not exist on the SD card. If a directory named **Dir1** is already present on the SD card, the application will **fail**.

File system layer uses:

- SDHC Driver to communicate to SD-Card

Building The Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/fs/sdhc_fat/firmware
-------------------------	---------------------------------------

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_V71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

- File System:
 - FAT Format for SDCARD
- SDHC Driver
- Timer System Service
- TC peripheral library

The following components used may vary based on the project configuration selected:

- Board Support Package (BSP)
- FreeRTOS

Other MHC Configurations:

- Enable XDMAC channels for SDHC Transmit/Receive

Hardware Setup

This section describes how to configure the supported hardware.

Description

- Project sam_e70_xult.X.
- Project sam_e70_xult_freertos.X.
 - Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
 - Hardware Setup**
 - Create a new JPEG file in the micro SD-Card with name **FILE_TOO_LONG_NAME_EXAMPLE_123.JPG**
 - Insert the micro SD-Card in the micro SD-Card slot of the Device.

3. Project sam_V71_xult_freertos.X.

- **Hardware Used**
 - SAM V71 Xplained Ultra Evaluation Kit
- **Hardware Setup**
 - Create a new JPEG file in the SD-Card with name **FILE_TOO_LONG_NAME_EXAMPLE_123.JPG**
 - Insert the SD Card in the SD Card slot of the Device.

Running The Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using MPLAB X IDE.
3. Insert the SD-Card in SD card Slot of the kit which has the file **FILE_TOO_LONG_NAME_EXAMPLE_123.JPG**
4. The LED is turned ON if there was no error during creating the directory and copying the file into it.

The following table provides the LED name:

Kit Name	LED Name
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 0

5. If LED is ON, Insert the SD-Card on to your host PC.
6. SD-Card should have the file **Dir1/FILE_TOO_LONG_NAME_EXAMPLE_123_1.JPG** and should match the original image.

sdspi_fat

This application shows an example of using the MPLAB Harmony File System to access and modify the contents of a SD card using the SDSPI driver.

Description

File System Operations on the SD-Card:

- The application opens a file named **FILE_TOO_LONG_NAME_EXAMPLE_123.JPG** on the SD card
- Reads the content of the file and creates a directory named **Dir1**
- Inside the directory, writes the copied content into another file **FILE_TOO_LONG_NAME_EXAMPLE_123_1.JPG**

The image file could be any arbitrary JPEG (image) file chosen by the user and then renamed to **FILE_TOO_LONG_NAME_EXAMPLE_123.JPG**.

The reason for choosing a JPEG file for test purposes is that the duplicate file, **FILE_TOO_LONG_NAME_EXAMPLE_123_1.JPG** created by the demonstration could be easily verified for correctness.

If the new file inside **Dir1** opens for viewing on the computer and matches to original image, the test is deemed to have passed. Otherwise, if the file does not open (i.e., is corrupted), the test will be considered to have failed.

Note:

Since the application creates a directory named **Dir1**, it is important that the a folder with the same name does not exist on the SD card. If a directory named **Dir1** is already present on the SD card, the application will **fail**.

File system layer uses:

- SDSPI Driver to communicate to SD-Card over SPI interface.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/fs/sdspi/sync/firmware
------------------	----------------------------------

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE:

Project Name	Description
sam_c21n_xpro_freertos.X	SAM C21N Xplained Pro Evaluation Kit + FreeRTOS
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This topic provides information on MHC configuration.

Description

Refer to the MHC project graph for the configurations of the various components used.

Components used:

The components used may vary based on the project configuration selected:

- File System
 - FAT Format for SDCARD
- SDSPI driver configured to synchronous mode.
- Time System Service.
- TC peripheral library.

The following components used may vary based on the project configuration selected:

- SPI or SERCOM peripheral library.
- Board Support Package (BSP)
- FreeRTOS

Other MHC Configurations:

- Configure the SD Card chip select pin in Pin Settings

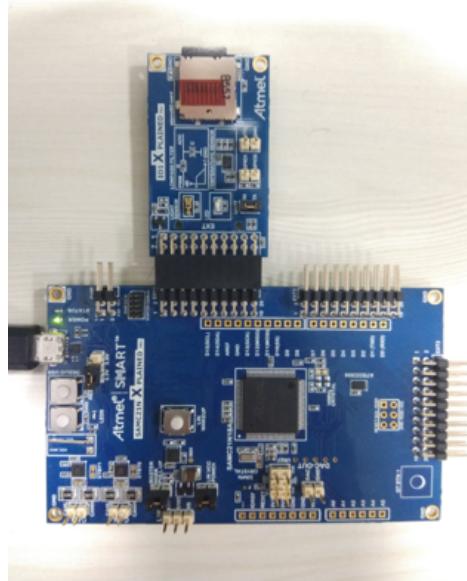
Hardware Setup

This section describes how to configure the supported hardware.

Description

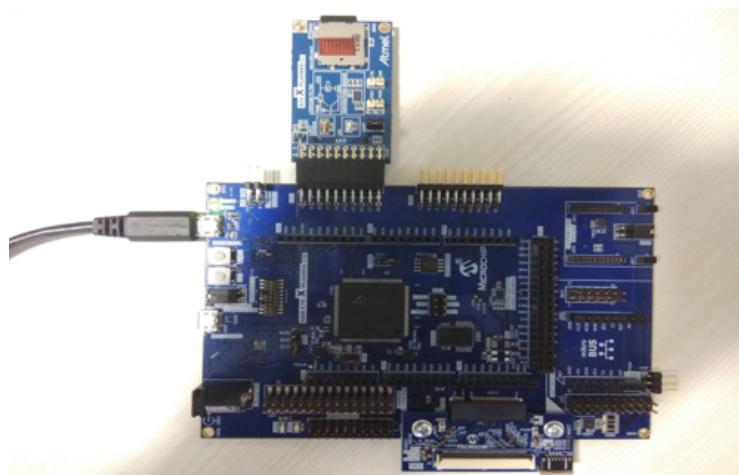
- Project sam_c21n_xpro_freertos.X.
 - Hardware Used

- SAM C21N Xplained Pro Evaluation Kit
- IO1 Xplained Pro Extension Kit
- **Hardware Setup**
 - Connect the [IO1 Xplained Pro Extension Kit](#) to the EXT1 header of the [SAM C21N Xplained Pro Evaluation Kit](#)
 - Create a new JPEG file in the micro SD-Card with name **FILE_TOO_LONG_NAME_EXAMPLE_123.JPG**
 - Insert the micro SD-Card in the micro SD-Card slot of the [IO1 Xplained Pro Extension Kit](#)



2. Project sam_e70_xult_freertos.X.

- **Hardware Used**
 - SAM E70 Xplained Ultra Evaluation Kit
 - [IO1 Xplained Pro Extension Kit](#)
- **Hardware Setup**
 - Connect the [IO1 Xplained Pro Extension Kit](#) to the EXT1 header of the SAM E70 Xplained Pro Evaluation Kit.
 - Create a new JPEG file in the micro SD-Card with name **FILE_TOO_LONG_NAME_EXAMPLE_123.JPG**
 - Insert the micro SD-Card in the micro SD-Card slot of the [IO1 Xplained Pro Extension Kit](#)



Running the Application

This section provides information on how to run the application using MPLAB X IDE.

Description

1. Connect a micro USB cable to the DEBUG port.
2. Build and Program the application using MPLAB X IDE.
3. Insert the micro SD-Card in micro SD card Slot of the kit which has the file **FILE_TOO_LONG_NAME_EXAMPLE_123.JPG**
4. The LED is turned ON if there was no error during creating the directory and copying the file into it.

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1

5. If LED is ON, Insert the SD-Card on to your host PC.
6. SD-Card should have the file **Dir1/FILE_TOO_LONG_NAME_EXAMPLE_123_1.JPG** and should match the original image.

RTOS Applications

This section provides help for the RTOS applications.

FreeRTOS Applications

This section provides descriptions of the FreeRTOS RTOS demonstrations.

basic_freertos

This example application blinks an LED on a starter kit to show the FreeRTOS threads that are running and to indicate status.

Description

This demonstration creates three tasks and a queue. Task1 sends message to Task2 and Task3 to unblock and toggle an LED. Then Task1 blocks itself for 200ms to allow other tasks to schedule and run.

- If Task2 has received expected value from Task1 then it toggles an LED and blocks itself for next "ulReceivedValue" amount of time
- If Task1 has received expected value from Task1 then it toggles an LED and blocks itself for next "ulReceivedValue" amount of time

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core\apps\rtos\freertos\basic_freertos\firmware

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE.

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_d20_xpro.X	SAM D20 Xplained Pro Evaluation Kit
sam_d21_xpro.X	SAM D21 Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit

MPLAB Harmony Configurations

This section provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components Used:

- FreeRTOS

The following components used may vary based on the project configuration selected:

- Board Support Package (BSP)

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_c21n_xpro.X.

- **Hardware Used:**
 - SAM C21N Xplained Pro Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup is required

2. Project sam_d20_xpro.X.

- **Hardware Used:**
 - SAM D20 Xplained Pro Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup is required

3. Project sam_d21_xult.X.

- **Hardware Used:**
 - SAM D21 Xplained Pro Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup is required

4. Project sam_e70_xult.X.

- **Hardware Used:**
 - SAM E70 Xplained Ultra Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup is required

5. Project sam_v71_xult.X.

- **Hardware Used:**
 - SAM V71 Xplained Ultra Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup is required

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB to the DEBUG port.
2. Build and program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED toggles on success i.e. each time when the Task1 or Task2 receives a valid message

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM D20 Xplained Pro Evaluation Kit	LED 0
SAM D21 Xplained Pro Evaluation Kit	LED 0
SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 0

task_notification_freertos

This example application is to illustrate the FreeRTOS Task Notification feature which is used as a light weight binary semaphore. This demonstration application toggles an LED for every 500ms.

Description

This demonstration creates two tasks that send notifications back and forth to each other.

- Task2 blocks to wait for Task1 to notify and will be blocked for 500ms
- Task1 sends a notification to Task2, bringing it out of the blocked state, toggles an LED
- Task1 blocks to wait for Task2 to notify
- Task2 sends notification to Task1, bringing it out of the blocked state

Above steps will be repeated. i.e. an LED toggles for every 500ms

Building the Application

This section provides information on how to build an application using MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core\apps\rtos\freertos\task_notification_freertos\firmware
------------------	---

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE.

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_v71_xult.X	SAM V71 Xplained Ultra Evaluation Kit

MPLAB Harmony Configurations

This section provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components Used:

- FreeRTOS

The following components used may vary based on the project configuration selected:

- Board Support Package (BSP)

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project sam_c21n_xpro.X.

- **Hardware Used:**
 - SAM C21N Xplained Pro Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup is required

2. Project sam_e70_xult.X.

- **Hardware Used:**
 - SAM E70 Xplained Ultra Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup is required

3. Project sam_v71_xult.X.

- **Hardware Used:**
 - SAM V71 Xplained Ultra Evaluation Kit
- **Hardware Setup:**
 - No special hardware setup is required

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB to the DEBUG port.
2. Build and program the application using the MPLAB X IDE.
3. The LED indicates the success or failure.
 - The LED toggles on success for every 500ms

The following table provides the LED name:

Kit Name	LED Name
SAM C21N Xplained Pro Evaluation Kit	LED 0
SAM D20 Xplained Pro Evaluation Kit	LED 0
SAM D21 Xplained Pro Evaluation Kit	LED 0

SAM E70 Xplained Ultra Evaluation Kit	LED 1
SAM V71 Xplained Ultra Evaluation Kit	LED 0

System Services Applications

This section provides help for the System Services applications.

Time System Service Applications

This section provides help for the System Service applications.

sys_time_multiclient

This example application demonstrates the multi-client system timer functionality.

Description

This application demonstrates timer functionality (with two clients to the Time System Service) by periodically printing a message on console every 2 seconds and blinking an LED every 1 second.

Delay and counter functionality is demonstrated on a switch press.

On a switch press, the application reads the current value of the 64 bit counter (say, count 1). It then starts a delay of 500 milliseconds and waits for the delay to expire. Once the delay has expired, the application again reads the current value of the 64 bit counter (say, count 2) and calculates the difference between the two counter values.

The difference count indicates the time spent for the delay and is printed on the console as,

Delay time = x ms

where x is the delay value and is equal to 500 milliseconds in the given example.

The application then starts a single shot timer of 100 milliseconds. When the single shot timer expires, a message is printed on the console that says "Single shot timer of 100 ms expired". Being a single shot timer, this message is printed only once on every switch press.

Building the Application

This section provides information on how to build an application using the MPLAB X IDE.

Description

The parent folder for all the MPLAB X projects for this application is given below:

Application Path	core/apps/system/time/sys_time_multiclient/firmware

To build the application, refer the following table and open the appropriate project file in the MPLAB X IDE.

Project Name	Description
sam_c21n_xpro.X	SAM C21N Xplained Pro Evaluation Kit
sam_d20_xpro.X	SAM D20 Xplained Pro Evaluation Kit
sam_d21_xpro.X	SAM D21 Xplained Pro Evaluation Kit
sam_e70_xult.X	SAM E70 Xplained Ultra Evaluation Kit
sam_e70_xult_freertos.X	SAM E70 Xplained Ultra Evaluation Kit + FreeRTOS
sam_v71_xult.X	SAM V71 Xplained Ultra Evaluation Kit
sam_v71_xult_freertos.X	SAM V71 Xplained Ultra Evaluation Kit + FreeRTOS

MPLAB Harmony Configuration

This section provides information on the MHC configurations.

Description

Refer to the MHC project graph for the configurations of various components used.

Components Used:

- Time System Service
- Timer peripheral library
- STDIO

The following components used may vary based on the project configuration selected.

- USART or SERCOM
- Board Support Package (BSP)
- FreeRTOS

Other MHC Settings:

- Configure pins for the USART or SERCOM peripheral in Pin Settings

Hardware Setup

This section describes how to configure the supported hardware.

Description

1. Project: `sam_c21n_xpro.X`.

- **Hardware Used:**
 - [SAM C21N Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - No special hardware setup is required

2. Project: `sam_d20_xpro.X`.

- **Hardware Used:**
 - [SAM D20 Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - No special hardware setup is required

3. Project: `sam_d21_xult.X`.

- **Hardware Used:**
 - [SAM D21 Xplained Pro Evaluation Kit](#)
- **Hardware Setup:**
 - No special hardware setup is required

4. Project: `sam_e70_xult.X`.

5. Project: `sam_e70_xult_freertos.X`.

- **Hardware Used:**
 - [SAM E70 Xplained Ultra Evaluation Kit](#)
- **Hardware Setup:**
 - No special hardware setup is required

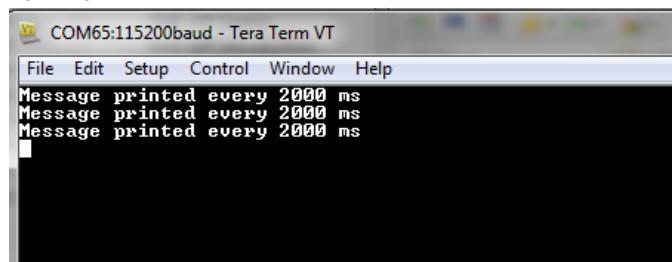
6. Project: **sam_v71_xult.X.**
7. Project: **sam_v71_xult_freertos.X.**
 - **Hardware Used:**
 - SAM V71 Xplained Ultra Evaluation Kit
 - **Hardware Setup:**
 - No special hardware setup is required

Running the Application

This section provides information on how to run an application using the MPLAB X IDE.

Description

1. Connect a micro USB to the DEBUG port.
2. Open Terminal application (Ex.:Tera Term) on the computer.
3. Configure the serial port settings as follows:
 - Baud : 115200
 - Data : 8 Bits
 - Parity : None
 - Stop : 1 Bit
 - Flow Control : None
4. Build and program the application using the MPLAB X IDE.
5. Observe the following message getting printed on the console every 2 seconds.



COM65:115200baud - Tera Term VT

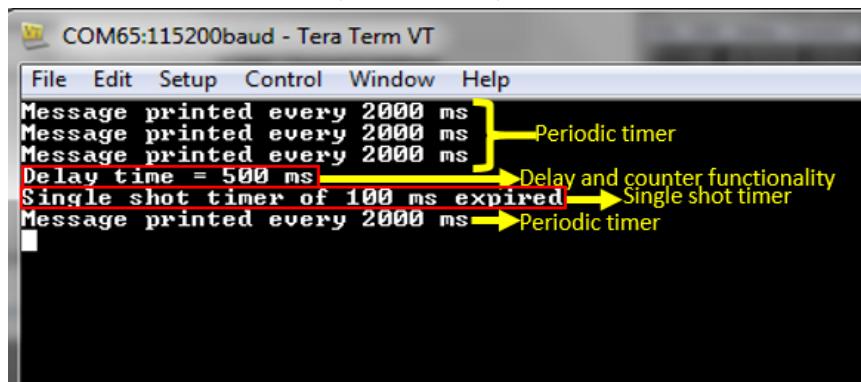
File Edit Setup Control Window Help

Message printed every 2000 ms

Message printed every 2000 ms

Message printed every 2000 ms

6. Press the switch and observe the following output on the terminal (highlighted in red box).
 - "Delay time = 500 ms" indicates the amount of time spent during the delay
 - "Single shot timer of 100 ms expired" is printed only once on every switch press



COM65:115200baud - Tera Term VT

File Edit Setup Control Window Help

Message printed every 2000 ms

Message printed every 2000 ms

Message printed every 2000 ms

Delay time = 500 ms

Single shot timer of 100 ms expired

Message printed every 2000 ms

Periodic timer

Delay and counter functionality

Single shot timer

Periodic timer

7. The LED indicates the success or failure.
 - The LED is toggles for every 1 second on success
- The following table provides the LED and Switch name:

Kit Name	LED Name	Switch Name
SAM C21N Xplained Pro Evaluation Kit	LED 0	SW0
SAM D20 Xplained Pro Evaluation Kit	LED 0	SW0
SAM D21 Xplained Pro Evaluation Kit	LED 0	SW0
SAM E70 Xplained Ultra Evaluation Kit	LED 1	SW1
SAM V71 Xplained Ultra Evaluation Kit	LED 0	SW0

Support

This section provides support information for MPLAB Harmony.

Using the Help

This topic contains general information that is useful to know to maximize using the MPLAB Harmony help.

Description

Help Formats

MPLAB Harmony Help is provided in three formats:

- Stand-alone HyperText Markup Language (HTML)
- Microsoft Compiled HTML Help (CHM)
- Adobe® Portable Document Format (PDF)

TIP! When using the MPLAB Harmony Help PDF, be sure to open the "bookmarks" if they are not already visible to assist in document navigation. See [Using the Help](#) for additional information.

Help File Locations

Each of these help files are included in the installation of MPLAB Harmony in the following locations:

- HTML - <install-dir>/doc/html/index.html
- CHM - <install-dir>/doc/help_harmony.chm
- PDF - <install-dir>/doc/help_harmony.pdf

Refer to [Help Features](#) for more information on using each output format.

Where to Begin With the Help

The help documentation provides a comprehensive source of information on how to use and understand MPLAB Harmony. However, it is not required to read the entire document before starting to work with MPLAB Harmony.

Prior to using MPLAB Harmony, it is recommended to review the Release Notes for any known issues. A PDF copy of the release notes is provided in the <install-dir>/doc folder of your installation.

New Users

For new users to MPLAB Harmony, it is best to follow the [Guided Tour](#) provided in *Volume I: Getting Started With MPLAB Harmony*.

Experienced Users

For experienced users already somewhat familiar with the MPLAB Harmony installation and online resources and, looking to jump right into a specific topic, follow the links provided in the table in *Volume 1: Getting Started With MPLAB Harmony Libraries and Applications* > [Guided Tour](#).

Trademarks

Provides information on trademarks used in this documentation.

Description

Trademarks

The Microchip name and logo, the Microchip logo, AnyRate, AVR, AVR logo, AVR Freaks, BeaconThings, BitCloud, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, Heldo, JukeBlox, KeeLoq, KeeLoq logo, Kleer, LANCheck, LINK MD, maXStylus, maXTouch, MediaLB, megaAVR, MOST, MOST logo, MPLAB, OptoLyzer, PIC, picoPower, PICSTART, PIC32 logo, Prochip Designer, QTouch, RightTouch, SAM-BA, SpyNIC, SST, SST Logo, SuperFlash, tinyAVR, UNI/O, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

ClockWorks, The Embedded Control Solutions Company, EtherSynch, Hyper Speed Control, HyperLight Load, IntelliMOS, mTouch, Precision Edge, and Quiet-Wire are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BodyCom, chipKIT, chipKIT logo, CodeGuard, CryptoAuthentication, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo,

Mindi, MiWi, motorBench, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PureSilicon, QMatrix, RightTouch logo, REAL ICE, Ripple Blocker, SAM-ICE, Serial Quad I/O, SMART-I.S., SQI, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

ARM and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and other countries.

All other trademarks mentioned herein are property of their respective companies.

© 2017, Microchip Technology Incorporated, All Rights Reserved.

Typographic Conventions

This topic describes the typographic conventions used in the MPLAB Harmony Help.

Description

The MPLAB Harmony Help uses the following typographic conventions:

Convention	Represents	Example
TIP!	Provides helpful information to assist the user.	
 Note:	Provides useful information to the user.	
Important!	Provides important information to the user.	
Warning	Warns the user of a potentially harmful issue.	
Followed by Green italicized text	Indicates a process step that is automated by the MPLAB Harmony Configurator (MHC).	
Italic Characters	Referenced documentation and emphasized text.	<ul style="list-style-type: none"> <i>MPLAB X IDE User's Guide</i> ...is the <i>only</i> option.
Initial Capitalization	<ul style="list-style-type: none"> A window A dialog A menu selection 	<ul style="list-style-type: none"> the Output window the SaveAs dialog the Enable Programmer menu
Quotation Marks	A field name in a window or dialog.	"Save project before build"
Italic text with right angle bracket	A menu path.	<i>File > Save</i>
Bold Characters	<ul style="list-style-type: none"> Topic headings A dialog button or user action, such as clicking an icon or selecting an option 	<ul style="list-style-type: none"> Prerequisites Click OK
Courier New text enclosed in angle brackets	A key on the keyboard.	Press <Ctrl><V> .
Courier New text	<ul style="list-style-type: none"> Sample source code File names File paths 	<ul style="list-style-type: none"> <code>#define START</code> <code>system_config.h</code> <code><install-dir>/apps/examples</code>
Square Brackets	Optional arguments.	<code>command [options] file [options]</code>

Curly Braces and Pipe Character	Choice of mutually exclusive arguments; an OR selection.	errorlevel {0 1}
---------------------------------	--	------------------

Recommended Reading

The following Microchip documents are available and recommended as supplemental reference resources.

Description

Device Data Sheets

Refer to the appropriate device data sheet for device-specific information and specifications.

Reference information found in these data sheets includes:

- Device memory maps
- Device pin out and packaging details
- Device electrical specifications
- List of peripherals included on the devices

To access this documentation, please visit, <http://www.microchip.com/pic32> and click **Documentation**. Then, expand **Data Sheets** to see the list of available documents.

MPLAB® XC32 C/C++ Compiler User's Guide (DS50001686)

This document details the use of Microchip's MPLAB XC32 Compiler for 32-bit microcontrollers to develop 32-bit applications. Please visit the Microchip website to access the latest version of this document.

MPLAB® X IDE User's Guide (DS50002027)

Consult this document for more information pertaining to the installation and implementation of the MPLAB X IDE software. Please visit the Microchip website to access the latest version of this document.

Documentation Feedback

This topic includes information on how to provide feedback on this documentation.

Description

Your valuable feedback can be provided to Microchip in several ways. Regardless of the method you use to provide feedback, please include the following information whenever possible:

- The Help platform you are viewing:
 - Adobe® Portable Document Format (PDF)
 - Windows® Compiled Help (CHM)
 - HyperText Markup Language (HTML)
- The title of the topic and the section in which it resides
- A clear description of the issue or improvement

How To Send Your Feedback

It is preferred that you use one of the following two methods to provide your feedback:

- Through the Documentation Feedback link, which is available in the header and footer of each topic when viewing compiled Help (CHM) or HTML Help
- By email at: docerrors@microchip.com

If either of the two previous methods are inconvenient, you may also provide your feedback by:

- Contacting your local Field Applications Engineer
- Contacting Customer Support at: <http://support.microchip.com>

Help Features

Describes the features available in the Help files provided in MPLAB Harmony.

CHM Help Features

Provides detailed information on the features available in CHM Help files.

Description

The MPLAB Harmony CHM files are located in the ./doc subfolder of the package it documents. For example, documentation on the MPLAB Harmony 3 Configurator is found at ./mhc/doc/help_mhc.chm and documentation on the MPLAB Harmony Graphics Library is found at ./gfx/doc/help_harmony_gfx.chm.

Help Icons

Several icons are provided in the interface of the Help, which aid in accessing the Help content.

Table 1: Help Icon Features

Help Icon	Description
	Use the Hide icon to turn off the left Help pane. Once the Hide icon is selected, it is replaced with the Show icon. Clicking the Show icon restores the left Help pane.
	Use the Locate icon to visually locate the Help topic you are viewing in the Contents. Clicking the Locate icon causes the current topic to be highlighted in blue in the Contents pane.
	Use the Back icon to move back through the previously viewed topics in the order in which they were viewed.
	Use the Forward icon to move forward through the previously viewed topics in the order in which they were viewed.
	Use the Home icon to return to the first topic in the Help.
	Use the Print icon to print the current topic or the selected heading and all subtopics.
	Use the Options icon to: <ul style="list-style-type: none"> • Hide tabs • Locate a topic • Go Back, Forward, and Home • Stop • Refresh • Set Internet Explorer options • Print topics • Turn Search Highlight Off and On

Topic Window

The Topic Window displays the current topic. In addition to the Help content, special links are provided in the upper portion of the window, as shown in Figure 2. Table 2 lists and describes the different links by their category

Figure 2: Help Links

Table 2: Help Links

Link Category	Description
Topic Path	The full path of the current topic is provided at the top and bottom of each topic, beginning with the top-level section name.
Support and Feedback Links:	<ul style="list-style-type: none"> • Documentation Feedback • Microchip Support Click this link to send feedback in the form of an email (see Note 1). Click this link to open the Microchip Support Web page.

Main Help Links:	<ul style="list-style-type: none"> Contents Index Home 	Click this link to open the Contents in the left pane. Click this link to open the Index in the left pane (see Note 2). Click this link to go to the initial Help topic (see Note 2).
Navigation Links:	<ul style="list-style-type: none"> Previous Up Next 	Click this link to go back to the previously viewed topic. Click this link to go to the parent section of the topic. Click this link to go to the next topic.

**Notes:**

1. To use the *Documentation Feedback* link, you must have an email system, such as Outlook configured. Clicking the link automatically opens a new email window and populates the recipient and subject lines.
2. The *Home* and *Index* links do not appear initially. Once you begin traversing the topics, they dynamically appear.

Tabs

The CHM Help provides four Tabbed windows: *Contents*, *Index*, *Search*, and *Favorites*.

Contents

The Contents tab displays the top-level topics/sections. Figure 3 shows the initial view when the CHM Help is first opened.

Figure 3: Initial Contents Tab View

As topics are explored, the information in the Contents tab dynamically updates. For example, by clicking **Prebuilt Libraries Help** and using the [Next](#) link in the current topic to traverse through this section, the collapsed section automatically expands and the current topic is highlighted in light gray, as shown in Figure 4.

Figure 4: Current Topic Highlighting

Index

Clicking the Index tab results in an alphabetic list of all Help index entries. Figure 5 shows the default Index interface.

Figure 5: Default Index Interface

- To locate a specific entry, enter the keyword in the *Type in the keyword to find:* box. As you type, the index list dynamically updates.
- To display the desired item in the list, select the item and click **Display**, or double-click the desired item. The related content appears in the Help window.

Search

Clicking the Search tab provides an efficient way to find specific information. Figure 6 shows the default Search interface.

Figure 6: Default Search Interface

- Enter the specific word or words in the *Type in the word(s) to search for:* box
- Clicking the drop-down arrow provides the list of previously searched words
- The right arrow provides Advanced Search options: AND, OR, NEAR, and NOT
- Located at the bottom left of the Search window, three options are provided to narrow-down your search. By default, *Match similar words* is selected. To reduce the number of returned words, clear this box and select *Search titles only*, which restricts the search to only the topic titles in the Help, as shown in Figure 7.

Figure 7: Search Titles Only

- The *Title* column provides the list of related topics
- The *Location* column lists in which Help system the topic was found (see **Note**)
- The *Rank* column determines to search result that most closely matches the specified word



Note: The *Location* column is automatically included in the CHM Help when the Advanced Search features are implemented and cannot be excluded. Its purpose is to provide the name of the Help system in which the topic is located for Help output that is generated from multiple sources. Since the MPLAB Harmony Help is contained within a single Help system, this information is the same for all searches. Do not confuse this column to mean the actual topic location.

Favorites

Use the Favorites tab to create a custom list of topics that you may want to repeatedly access. Figure 8 shows the default Favorites interface.

Figure 8: Default Favorites Interface

- The title of the current topic is shown in the *Current topic:* box.
- Click **Add** to add the topic to the *Topics:* list, as shown in Figure 9.
- Click **Display** to view the selected topic.
- Click **Remove** to remove the selected topic from the list of favorites.

Figure 9: Adding a Favorite Topic

HTML Help Features

Provides detailed information on the features available in the stand-alone HTML Help.

Description

The HTML Help output for MPLAB Harmony has two purposes. First, it can be used as "stand-alone" Help. Second, the HTML files are used by the MPLAB Harmony Configurator (MHC) when using MHC in MPLAB X IDE.

Stand-alone HTML Help

The MPLAB Harmony index.html file that is the root for all HTML help is located in the ./doc subfolder of the package it documents. For example, documentation on the MPLAB Harmony 3 Configurator is found at ./mhc/doc/index.html and documentation on the MPLAB Harmony Graphics Library is found at ./gfx/doc/index.html.

To use the HTML Help in a "stand-alone" manner, open the file index.html in your browser of choice. Click **Allow blocked content** if a message appears regarding ActiveX controls.

The following links are provided:

- *Table of Contents* - Located at the top left, clicking this link opens the Table of Contents in the right frame
- *Topic Path* - At the top and bottom of each topic, the full path to the current topic is listed
- *Microchip Logo* - Clicking this image opens a new browser tab and displays the Microchip website (www.microchip.com)
- *Contents* - The Contents topic is a static file, which displays and lists the major sections available in the Help in the left frame. Due to a restriction with the Help browser used by the MHC, a dynamic Contents topic cannot be used.
- *Home* - This link returns to the Introduction topic (see **Note 1**)
- *Previous* and *Next* navigation links - Use these links to traverse through the Help topics
- *Documentation Feedback* - Use this link to provide feedback in the form of an email (see **Note 2**)
- *Microchip Support* - Use this link to open the Support page of the Microchip website



Notes:

1. The *Home* link does not appear initially. Once you begin traversing the topics, it dynamically appears.
2. To use the *Documentation Feedback* link, you must have an email system such as Outlook configured. Clicking the link automatically opens a new email message and populates the recipient and subject lines.

PDF Help Features

Provides detailed information on the features available in the PDF version of the Help.

Description

The MPLAB Harmony Help provided in Portable Document Format (PDF) provides many useful features. By default, PDF

bookmarks should be visible when opening the file. If PDF bookmarks are not visible, click the PDF Bookmark icon, which is located near the top of the left navigation pane or by selecting *View > Show/Hide > Navigation Panes > Bookmarks*.

The MPLAB Harmony PDF files are located in the *./doc* subfolder of the package it documents. For example, documentation on the MPLAB Harmony 3 Configurator is found at *./mhc/doc/help_mhc.pdf* and documentation on the MPLAB Harmony Graphics Library is found at *./gfx/doc/help_harmony_gfx.pdf*.

To make full use of the PDF features, it is recommended that Adobe products be used to view the documentation (see **Note**).

Help on how to use the PDF features is available through your copy of Acrobat (or Acrobat Reader) by clicking **Help** in the main menu.



Note: The MPLAB Harmony Help PDF files can be viewed using a PDF viewer or reader that is compatible with Adobe PDF Version 7.0 or later.

Microchip Website

This topic provides general information on the Microchip website.

Description

The Microchip website can be accessed online at: <http://www.microchip.com>

Accessible by most Internet browsers, the following information is available:

Product Support

- Data sheets
- Silicon errata
- Application notes and sample programs
- Design resources
- User's guides
- Hardware support documents
- Latest software releases and archived software

General Technical Support

- Frequently Asked Questions (FAQs)
- Technical support requests
- Online discussion groups
- Microchip consultant program member listings

Business of Microchip

- Product selector and ordering guides
- Latest Microchip press releases
- Listings of seminars and events
- Listings of Microchip sales offices, distributors, and factory representatives

Microchip Forums

This topic provides information on the Microchip Web Forums.

Description

The Microchip Web Forums can be accessed online at: <http://www.microchip.com/forums>

Microchip provides additional online support via our web forums.

The Development Tools Forum is where the MPLAB Harmony forum discussion group is located. This forum handles questions and discussions concerning the MPLAB Harmony Integrated Software Framework and all associated libraries and components.

Additional Development Tool discussion groups include, but are not limited to:

- MPLAB X IDE - This forum handles questions and discussions concerning the released versions of the MPLAB X Integrated Development Environment (IDE)
- MPLAB XC32 - This forum handles questions and discussions concerning Microchip's 32-bit compilers, assemblers, linkers, and related tools for PIC32 microcontrollers
- Tips and Tricks - This forum provides shortcuts and quick workarounds for Microchip's development tools
- FAQs - This forum includes Frequently Asked Questions

Additional forums are also available.

Customer Support

This topic provides information for obtaining support from Microchip.

Description

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office (see [Contact Microchip Technology](#) to locate your local sales office)
- Field Application Engineer (FAE)
- Technical Support (<http://support.microchip.com>)

Contact Microchip Technology

Worldwide sales and service contact information for Microchip Technology Inc.

Description

Please visit the following Microchip Web page for contact information: <http://www.microchip.com/about-us/contact-us>

Glossary

This topic contains a glossary of general MPLAB Harmony terms.

Description

Glossary of MPLAB Harmony Terms

Term	Definition
Application	One or more application modules define the overall behavior of a MPLAB Harmony system. Applications are either demonstrations or examples provided with the installation or are implemented by you, using MPLAB Harmony libraries to accomplish a desired task.
Client	A client module is any module that uses the services (calls the interface functions) of another module.
Configuration	A MPLAB Harmony configuration consists of static definitions (C language <code>#define</code> statements), executable source code, and other definitions in a set of files that are necessary to create a working MPLAB Harmony system. (See the System Configurations section for additional information.)
Configuration Options	Configuration options are the specific set of <code>#define</code> statements that are required by any specific MPLAB Harmony library to specify certain parameters (such as buffer sizes, minimum and maximum values, etc.) build that library. Configuration options are defined in the <code>system_config.h</code> system-wide configuration header.
Driver	A "driver" (or device driver) is a MPLAB Harmony software module designed to control and provide access to a specific peripheral, either built into or external to the microcontroller.
Driver Index	Dynamic MPLAB Harmony drivers (and other dynamic modules) can manage the more than one instance of the peripheral (and other resources) that they control. The "driver index" is a static index number (0, 1, 2,...) that identifies which instance of the driver is to be used.  Note: The driver index is not necessarily identical to the peripheral index. The association between these two is made when the driver is initialized.
Driver Instance	An instance of a driver (or other module) consists of a complete set of the memory (and other resources) controlled by the driver's code. Selection of which set of resources to control is made using a driver index.  Note: Even though there may be multiple instances of the resources managed by a dynamic driver, there is only ever one instance of the actual object code. However, static drivers always maintain a 1:1 relationship between resource and code instances.
Framework	The MPLAB Harmony framework consists of a set of libraries (and the rules and conventions used to create those libraries) that can be used to create MPLAB Harmony systems.

Handle	A handle is a value that allows one software module to "hold" onto a specific instance of some object owned by another software module (analogous to the way a valet holds the handle of a suitcase), creating a link between the two software modules. A handle is an "opaque" value, meaning that the "client" module (the module that receives and holds the handle) must not attempt to interpret the contents or meaning of the handle value. The value of the handle is only meaningful to the "server" module (the module that provides the handle). Internal to the server module, the handle may represent a memory address or it may represent a zero-based index or any other value, as required by the "server" module to identify the "object" to which the client is linked by the handle.
Initialization Overrides	Initialization overrides are configuration options that can be defined to statically override (at build time) parameters that are normally passed into the "Initialize" function of a driver or other MPLAB Harmony module. This mechanism allows you to statically initialize a module, instead of dynamically initializing the module.
Interface	The interface to a module is the set of functions, data types, and other definitions that must be used to interact with that module.
Middleware	The term "middleware" is used to describe any software that fits between the application and the device drivers within a MPLAB Harmony system. This term is used to describe libraries that use drivers to access a peripheral, and then implement communication protocols (such as TCP/IP, USB protocols, and graphics image processing), as well as other more complex processing, which is required to use certain peripherals, but is not actually part of controlling the peripheral itself.
Module	A MPLAB Harmony software module is a closely related group of functions controlling a related set of resources (memory and registers) that can be initialized and maintained by the system. Most MPLAB Harmony modules provide an interface for client interaction. However, "headless" modules with no interface are possible.
Peripheral Index	A peripheral index is a static label (usually an C language "enum" value) that is used to identify a specific instance of a peripheral.  Note: Unlike a driver index, which always starts at '0', a peripheral index may be internally represented as any number, letter, or even a base address and the user should not rely on the value itself, but only the label.
Peripheral Instance	An instance of a peripheral is a complete set of the registers (and internal physical resources) necessary to provide the core functionality of a given type of peripheral (either built into or external to the microcontroller).  Note: A specific peripheral instance is identified using a peripheral index.
System	A MPLAB Harmony system is a complete set of libraries, applications, and configuration items loaded and executing on a specific hardware platform (microcontroller, board, and external peripherals) or the source items necessary to build such a system.  Note: Since a system can multiple configurations, one MPLAB Harmony project may support multiple systems through multiple supported configurations. See the demonstration applications included in the installation for examples.
System Service	A system service is a MPLAB Harmony module that provides access to and control of common system resources (memory and registers) with which other modules (drivers, middleware, libraries and application) may interact.  Note: System services, much like drivers, manage sharing of resources so as to avoid conflicts between modules that would otherwise occur if each module attempted to manage the share resource itself. But, unlike drivers, system services do not normally need to be "opened" to use them.

Index

- - _OSAL_H macro 492
 - _promptStr macro 268
 - _SYS_DEBUG_MESSAGE macro 307
 - _SYS_DEBUG_PRINT macro 308
 - _SYS_FS_VOLUME_PROPERTY structure 404
 - _SYS_TIME_INIT structure 461
- A**
 - Abstraction Model 36, 57, 73, 94, 116, 158, 184, 217, 234, 259, 285, 298, 311, 323, 409, 424, 440, 471
 - Command Processor System Service Library 259
 - Console System Service Library 285
 - Debug System Service Library 298
 - DMA System Service Library 311
 - Interrupt System Service Library 409
 - OSAL Library 471
 - Ports System Service Library 424
 - SPI Driver Library 184
 - USART Driver Library 234
 - Application Interaction 324
 - Applications Help 495
 - Asynchronous Mode 184
 - AT24 Driver Applications 495
 - AT24 Driver Library Help 36
 - at24_eeprom_read_write 495
 - AT25 Driver Applications 498
 - AT25 Driver Library Help 56
 - at25_eeprom_read_write 498
- B**
 - basic_freertos 559
 - Building the Application 495, 498, 502, 504, 506, 509, 517, 520, 521, 523, 526, 527, 529, 533, 535, 537, 539, 548, 557, 559, 561, 563
 - Building The Application 511, 513, 515, 531, 542, 544, 546, 551, 553, 555
 - Building the Library 38, 58, 75, 105, 120, 160, 187, 219, 237, 260, 287, 300, 321, 328, 413, 442, 476
 - Command Processor System Service Library 260
 - Console System Service Library 287
 - Debug System Service Library 300
 - DMA System Service Library 321
 - File System Service Library 328
 - I2C Driver Library 75
 - I2S Driver Library 105
 - Interrupt System Service Library 413
 - OSAL Library 476
 - SPI Driver Library 187
 - USART Driver Library 237
- C**
 - Channel Setup and Management 314
 - CHM Help Features 570
 - Client Access 96
 - Client Operations - Buffered 97
 - Client Operations - Non-buffered 100
 - Command Processor System Service Library 259
- COMMAND_HISTORY_DEPTH macro 268
- Configuring MHC 104
- Configuring the Library 37, 58, 74, 104, 118, 159, 187, 218, 235, 260, 286, 299, 321, 326, 411, 424, 441, 475
 - Command Processor System Service Library 260
 - Console System Service Library 286
 - Debug System Service Library 299
 - DMA System Service Library 321
 - File System Service Library 326
 - Interrupt System Service Library 411
 - OSAL Library 475
 - Ports System Service Library 424
 - SPI Driver Library 187
 - USART Driver Library 235
- Console System Service Library 284
- Contact Microchip Technology 574
- Critical Section Operation 474
- Critical Sections 410
- Customer Support 574
- D**
 - Debug System Service Library 298
 - Developer Help 4
 - Direct Memory Access (DMA) System Service Library 310
 - Documentation 8
 - Documentation Feedback 569
 - Driver Applications 495
 - Driver Libraries Help 23
 - Driver Library Overview 23
 - driver.h 35
 - driver_common.h 114
 - Drivers 23
 - drv_at24.h 54
 - DRV_AT24_Close function 42
 - drv_at24_definitions.h 55
 - DRV_AT24_EVENT_HANDLER type 49
 - DRV_AT24_EventHandlerSet function 43
 - DRV_AT24_GEOMETRY structure 50
 - DRV_AT24_GeometryGet function 48
 - DRV_AT24_I2C_ERROR enumeration 52
 - DRV_AT24_INIT structure 52
 - DRV_AT24_Initialize function 39
 - DRV_AT24_Open function 41
 - DRV_AT24_PageWrite function 45
 - DRV_AT24_PLIB_CALLBACK type 53
 - DRV_AT24_PLIB_CALLBACK_REGISTER type 51
 - DRV_AT24_PLIB_ERROR_GET type 51
 - DRV_AT24_PLIB_INTERFACE structure 54
 - DRV_AT24_PLIB_IS_BUSY type 52
 - DRV_AT24_PLIB_READ type 52
 - DRV_AT24_PLIB_WRITE type 53
 - DRV_AT24_PLIB_WRITE_READ type 53
 - DRV_AT24_Read function 46
 - DRV_AT24_Status function 40
 - DRV_AT24_TRANSFER_STATUS enumeration 51
 - DRV_AT24_TransferStatusGet function 47
 - DRV_AT24_Write function 44
 - drv_at25.h 71

DRV_AT25_Close function 62
 DRV_AT25_EVENT_HANDLER type 70
 DRV_AT25_EventHandlerSet function 63
 DRV_AT25_GEOMETRY structure 69
 DRV_AT25_GeometryGet function 68
 DRV_AT25_Initialize function 59
 DRV_AT25_Open function 61
 DRV_AT25_PageWrite function 66
 DRV_AT25_Read function 64
 DRV_AT25_Status function 60
 DRV_AT25_TRANSFER_STATUS enumeration 70
 DRV_AT25_TransferStatusGet function 67
 DRV_AT25_Write function 65
 DRV_BAUDSET type 109
 DRV_CLIENT_STATUS enumeration 107
 DRV_HANDLE type 108
 DRV_HANDLE_INVALID macro 112
 drv_i2c.h 93
 DRV_I2C_Close function 79
 DRV_I2C_ErrorGet function 89
 DRV_I2C_Initialize function 76
 DRV_I2C_Open function 78
 DRV_I2C_ReadTransfer function 81
 DRV_I2C_ReadTransferAdd function 82
 DRV_I2C_Status function 77
 DRV_I2C_TRANSFER_EVENT enumeration 90
 DRV_I2C_TRANSFER_EVENT_HANDLER type 91
 DRV_I2C_TRANSFER_HANDLE type 90
 DRV_I2C_TRANSFER_HANDLE_INVALID macro 92
 DRV_I2C_TransferEventHandlerSet function 79
 DRV_I2C_TransferStatusGet function 88
 DRV_I2C_WriteReadTransfer function 86
 DRV_I2C_WriteReadTransferAdd function 87
 DRV_I2C_WriteTransfer function 83
 DRV_I2C_WriteTransferAdd function 84
 drv_i2s.h 114
 drv_i2s_definitions.h 114
 DRV_I2S_DMA_WIDTH enumeration 109
 DRV_I2S_ERROR enumeration 111
 DRV_I2S_INIT structure 108
 DRV_I2S_LRCLK_GET type 113
 DRV_I2S_PLIB_INTERFACE structure 109
 DRV_IO_BUFFER_TYPES enumeration 110
 DRV_IO_INTENT enumeration 110
 DRV_IO_ISBLOCKING macro 207
 DRV_IO_ISEXCLUSIVE macro 207
 DRV_IO_ISNONBLOCKING macro 208
 drv_memory.h 155
 DRV_MEMORY_AddressGet function 144
 DRV_MEMORY_AsyncErase function 128
 DRV_MEMORY_AsyncEraseWrite function 129
 DRV_MEMORY_AsyncRead function 131
 DRV_MEMORY_AsyncWrite function 133
 DRV_MEMORY_Close function 125
 DRV_MEMORY_COMMAND_HANDLE type 147
 DRV_MEMORY_COMMAND_HANDLE_INVALID macro 148
 DRV_MEMORY_COMMAND_STATUS enumeration 148
 DRV_MEMORY_CommandStatusGet function 142
 drv_memory_definitions.h 156
 DRV_MEMORY_DEVICE_CLOSE type 150
 DRV_MEMORY_DEVICE_EVENT_HANDLER_SET type 151
 DRV_MEMORY_DEVICE_GEOMETRY_GET type 151
 DRV_MEMORY_DEVICE_INTERFACE structure 149
 DRV_MEMORY_DEVICE_OPEN type 149
 DRV_MEMORY_DEVICE_PAGE_WRITE type 150
 DRV_MEMORY_DEVICE_READ type 150
 DRV_MEMORY_DEVICE_SECTOR_ERASE type 150
 DRV_MEMORY_DEVICE_STATUS type 150
 DRV_MEMORY_DEVICE_TRANSFER_STATUS_GET type 151
 DRV_MEMORY_Erase function 136
 DRV_MEMORY_EraseWrite function 138
 DRV_MEMORY_EVENT enumeration 151
 DRV_MEMORY_EVENT_HANDLER type 152
 DRV_MEMORY_GeometryGet function 145
 DRV_MEMORY_INIT structure 152
 DRV_MEMORY_Initialize function 121
 DRV_MEMORY_IsAttached function 146
 DRV_MEMORY_IsWriteProtected function 147
 DRV_MEMORY_Open function 124
 DRV_MEMORY_Read function 140
 DRV_MEMORY_Status function 123
 DRV_MEMORY_SyncErase function 135
 DRV_MEMORY_SyncEraseWrite function 136
 DRV_MEMORY_SyncRead function 138
 DRV_MEMORY_SyncWrite function 140
 DRV_MEMORY_Tasks function 124
 DRV_MEMORY_TRANSFER_HANDLE type 153
 DRV_MEMORY_TransferHandlerSet function 126
 DRV_MEMORY_TransferStatusGet function 143
 DRV_MEMORY_Write function 142
 drv_sdhc.h 182
 DRV_SDHC_AsyncRead function 170
 DRV_SDHC_AsyncWrite function 171
 DRV_SDHC_BUS_WIDTH enumeration 176
 DRV_SDHC_Close function 166
 DRV_SDHC_COMMAND_HANDLE type 176
 DRV_SDHC_COMMAND_HANDLE_INVALID macro 180
 DRV_SDHC_COMMAND_STATUS enumeration 177
 DRV_SDHC_CommandStatusGet function 173
 DRV_SDHC_Deinitialize function 163
 DRV_SDHC_EVENT enumeration 177
 DRV_SDHC_EVENT_HANDLER type 178
 DRV_SDHC_EventHandlerSet function 167
 DRV_SDHC_GeometryGet function 174
 DRV_SDHC_INDEX_0 macro 180
 DRV_SDHC_INDEX_COUNT macro 181
 DRV_SDHC_INIT structure 179
 DRV_SDHC_Initialize function 161
 DRV_SDHC_IsAttached function 175
 DRV_SDHC_IsWriteProtected function 175
 DRV_SDHC_Open function 165
 DRV_SDHC_Reinitialize function 162
 DRV_SDHC_SPEED_MODE enumeration 179
 DRV_SDHC_Status function 168
 DRV_SDHC_Tasks function 164
 drv_spi.h 216

DRV_SPI_Close function 192
 DRV_SPI_Initialize function 189
 DRV_SPI_Open function 191
 DRV_SPI_ReadTransfer function 196
 DRV_SPI_ReadTransferAdd function 199
 DRV_SPI_Status function 191
 DRV_SPI_TRANSFER_EVENT enumeration 204
 DRV_SPI_TRANSFER_EVENT_HANDLER type 205
 DRV_SPI_TRANSFER_HANDLE type 205
 DRV_SPI_TRANSFER_HANDLE_INVALID macro 207
 DRV_SPI_TransferEventHandlerSet function 194
 DRV_SPI_TransferSetup function 193
 DRV_SPI_TransferStatusGet function 203
 DRV_SPI_WriteReadTransfer function 198
 DRV_SPI_WriteReadTransferAdd function 202
 DRV_SPI_WriteTransfer function 197
 DRV_SPI_WriteTransferAdd function 200
 drv_sst26.h 233
 DRV_SST26_BulkErase function 225
 DRV_SST26_ChipErase function 226
 DRV_SST26_Close function 222
 DRV_SST26_GEOMETRY structure 232
 DRV_SST26_GeometryGet function 231
 DRV_SST26_Initialize function 220
 DRV_SST26_Open function 221
 DRV_SST26_PageWrite function 227
 DRV_SST26_Read function 228
 DRV_SST26_ReadJedecId function 223
 DRV_SST26_SectorErase function 224
 DRV_SST26_Status function 229
 DRV_SST26_TRANSFER_STATUS enumeration 232
 DRV_SST26_TransferStatusGet function 230
 DRV_SST26_UnlockFlash function 223
 drv_usart.h 257
 DRV_USART_BUFFER_EVENT enumeration 254
 DRV_USART_BUFFER_EVENT_HANDLER type 255
 DRV_USART_BUFFER_HANDLE type 254
 DRV_USART_BUFFER_HANDLE_INVALID macro 256
 DRV_USART_BufferCompletedBytesGet function 244
 DRV_USART_BufferEventHandlerSet function 243
 DRV_USART_BufferStatusGet function 246
 DRV_USART_Close function 242
 DRV_USART_ERROR type 256
 DRV_USART_ErrorGet function 253
 DRV_USART_INIT type 254
 DRV_USART_Initialize function 238
 DRV_USART_Open function 241
 DRV_USART_ReadBuffer function 247
 DRV_USART_ReadBufferAdd function 248
 DRV_USART_ReadQueuePurge function 249
 DRV_USART_SERIAL_SETUP type 257
 DRV_USART_SerialSetup function 240
 DRV_USART_Status function 239
 DRV_USART_WriteBuffer function 250
 DRV_USART_WriteBufferAdd function 251
 DRV_USART_WriteQueuePurge function 252
 Dynamic Configuration 413

E

ESC_SEQ_SIZE macro 268

F

FAT_FS_MAX_LFN macro 391
 FAT_FS_MAX_SS macro 392
 FAT_FS_USE_LFN macro 392
 File System Applications 541
 File System Service Library Help 322
 Files 35, 54, 71, 92, 113, 155, 181, 215, 233, 257, 282, 297, 309, 322, 405, 423, 424, 437, 467, 492
 Command Processor System Service Library 282
 Console System Service Library 297
 Debug System Service Library 309
 DMA System Service Library 322
 File System Service Library 405
 Interrupt System Service Library 423
 OSAL Library 492
 Ports System Service Library 424
 SPI Driver Library 215
 System Service Library Overview 437
 USART Driver Library 257

FreeRTOS Applications 559
G
 Global Control and Status 319
 Glossary 574
 Guided Tour 3
H
 Hardware Setup 496, 499, 502, 504, 507, 509, 512, 514, 516, 518, 521, 522, 524, 526, 528, 530, 532, 533, 536, 538, 540, 543, 545, 547, 549, 552, 553, 555, 557, 560, 562, 564
 Help Features 570
 help_osal.h 493
 How the Library Works 37, 57, 73, 95, 117, 158, 218, 235, 259, 285, 299, 312, 324, 409, 440, 473
 Command Processor System Service Library 259
 Console System Service Library 285
 Debug System Service Library 299
 DMA System Service Library 312
 Interrupt System Service Library 409
 OSAL Library 473
 USART Driver Library 235

HTML Help Features 572

I

I2C Driver Applications 501
 I2C Driver Library Help 72
 i2c_eeprom 501, 506
 i2c_multi_slave 503, 508
 I2S Driver Library Help 93
 Initialization and Tasks 312
 Installation 6
 INT_SOURCE type 422
 Interrupt System Service Library 408
 Interrupt System Setup 410
 Introduction 2, 36, 56, 72, 94, 115, 157, 183, 217, 234, 259, 284, 298, 310, 322, 408, 424, 425, 439, 470

L

Library Interface 35, 38, 59, 75, 106, 120, 160, 188, 219, 237, 260, 287, 300, 322, 329, 414, 424, 430, 442, 477
 Command Processor System Service Library 260
 Console System Service Library 287
 Debug System Service Library 300
 DMA System Service Library 322
 File System Service Library 329
 Interrupt System Service Library 414
 OSAL Library 477
 Ports System Service Library 424
 SPI Driver Library 188
 System Service Library 430
 USART Driver Library 237
 Library Overview 95, 312, 472
 DMA System Service Library 312
 File System Service Library 324
 OSAL Library 472
 LINE_TERM macro 268

M

MAIN_RETURN macro 210
 MAIN_RETURN_CODE macro 210
 MAIN_RETURN_CODES enumeration 112
 MAX_CMD_ARGS macro 269
 MAX_CMD_GROUP macro 269
 Memory Driver Applications 511
 Memory Driver Library Help 115
 Memory Operation 475
 Memory to Memory Transfer 319
 MEMORY_DEVICE_GEOMETRY structure 154
 MEMORY_DEVICE_TRANSFER_STATUS enumeration 154
 Microchip Forums 573
 Microchip Website 573
 MPLAB Harmony Configuration 495, 498, 502, 504, 507, 509, 512, 513, 515, 517, 520, 522, 524, 526, 527, 529, 531, 542, 544, 546, 548, 551, 553, 555, 557, 564
 MPLAB Harmony Configurations 533, 535, 538, 540, 560, 562
 MPLAB Harmony Module System Interface 426
 Mutex Operation 474

N

nvm_fat 541
 nvm_mpfs 544
 nvm_sdhc_fat_multi_disk 546
 nvm_sdspi_fat_multi_disk 548
 nvm_sst26_fat 550
 nvm_sst26_read_write 511, 513

O

Obtaining System Version Information 429
 Online Discussion Forum 5
 Opening a Driver 29
 OSAL Library Help 470
 OSAL Operation 475
 osal.h 493
 OSAL_CRIT_Enter function 486
 OSAL_CRIT_Leave function 486
 OSAL_CRIT_TYPE enumeration 490

OSAL_CRIT_TYPE_HIGH enumeration member 490
 OSAL_CRIT_TYPE_LOW enumeration member 490
 OSAL_Free function 488
 OSAL_Initialize function 489
 OSAL_Malloc function 487
 OSAL_MUTEX_Create function 483
 OSAL_MUTEX_DECLARE macro 492
 OSAL_MUTEX_Delete function 483
 OSAL_MUTEX_Lock function 484
 OSAL_MUTEX_Unlock function 485
 OSAL_Name function 489
 OSAL_RESULT enumeration 491
 OSAL_RESULT_FALSE enumeration member 491
 OSAL_RESULT_NOT_IMPLEMENTED enumeration member 491
 OSAL_RESULT_TRUE enumeration member 491
 OSAL_SEM_Create function 478
 OSAL_SEM_DECLARE macro 491
 OSAL_SEM_Delete function 479
 OSAL_SEM_GetCount function 482
 OSAL_SEM_Pend function 479
 OSAL_SEM_Post function 480
 OSAL_SEM_PostISR function 481
 OSAL_SEM_TYPE enumeration 491
 OSAL_SEM_TYPE_BINARY enumeration member 491
 OSAL_SEM_TYPE_COUNTING enumeration member 491
 Other Configuration-specific Files 22

P

PDF Help Features 572
 Ports System Service Library 423

Q

qspi_sst26_fat 552

R

Recommended Reading 569
 RTOS Applications 559
 Running the Application 497, 500, 503, 506, 508, 510, 519, 521, 523, 525, 527, 528, 530, 534, 536, 538, 541, 550, 558, 561, 562, 565
 Running The Application 512, 514, 516, 532, 543, 545, 547, 552, 554, 556

S

SD Card Host Controller Library 157
 SDHC Driver Applications 515
 SDHC_DETECTION_LOGIC enumeration 180
 sdhc_fat 554
 SDHC_MAX_LIMIT macro 181
 sdhc_read_write 515
 SDSPI Driver Applications 517
 sdspi_fat 556
 sdspi_read_write 517
 Semaphores 473
 Source Interrupt Management 411
 SPI Driver Applications 519
 SPI Driver Library Help 183
 spi_multi_instance 520, 525
 spi_multi_slave 521, 527
 spi_self_loopback_multi_client 523, 529
 SST26 Driver Applications 531

SST26 Driver Library Help 216
sst26_flash_read_write 531
Static Configuration 412
STDERR_FILENO macro 294
STDIN_FILENO macro 294
STDOUT_FILENO macro 294
Support 567
Synchronous mode 186
SYS_ASSERT macro 211
SYS_CMD_ADDGRP function 262
SYS_CMD_API structure 270
SYS_CMD_BUFFER_DMA_READY macro 276
SYS_CMD_CallbackFunction type 279
SYS_CMD_CONSOLE_IO_PARAM enumeration 280
SYS_CMD_DATA_RDY_FNC type 271
SYS_CMD_DELETE function 263
SYS_CMD_DESCRIPTOR structure 271
SYS_CMD_DESCRIPTOR_TABLE structure 272
SYS_CMD_DEVICE_LIST structure 272
SYS_CMD_DEVICE_MAX_INSTANCES macro 281
SYS_CMD_DEVICE_NODE structure 270
SYS_CMD_EVENT enumeration 280
SYS_CMD_FNC type 273
SYS_CMD_GETC_FNC type 273
SYS_CMD_HANDLE type 273
SYS_CMD_INIT structure 280
SYS_CMD_INIT_DATA structure 274
SYS_CMD_Initialize function 265
SYS_CMD_MAX_LENGTH macro 269
SYS_CMD_MESSAGE function 263
SYS_CMD_MESSAGE macro 281
SYS_CMD_MSG_FNC type 274
SYS_CMD_PRINT function 263
SYS_CMD_PRINT macro 281
SYS_CMD_PRINT_FNC type 274
SYS_CMD_PUTC_FNC type 275
SYS_CMD_READ_BUFFER_SIZE macro 270
SYS_CMD_READADC_FNC type 275
SYS_CMD_READY_TO_READ function 264
SYS_CMD_READY_TO_WRITE function 264
SYS_CMD_RegisterCallback function 266
SYS_CMD_STATE enumeration 275
SYS_CMD_Tasks function 265
SYS_CMDIO_ADD function 267
SYS_CMDIO_GET_HANDLE function 267
sys_command.h 282
sys_console.h 297
SYS_CONSOLE_CALLBACK type 296
SYS_CONSOLE_EVENT enumeration 296
SYS_CONSOLE_Flush function 293
SYS_CONSOLE_INDEX_0 macro 294
SYS_CONSOLE_INDEX_1 macro 295
SYS_CONSOLE_INDEX_2 macro 295
SYS_CONSOLE_INDEX_3 macro 295
SYS_CONSOLE_Initialize function 288
SYS_CONSOLE_MESSAGE macro 295
SYS_CONSOLE_PRINT macro 295
SYS_CONSOLE_Read function 290
SYS_CONSOLE_RegisterCallback function 292
SYS_CONSOLE_Status function 289
SYS_CONSOLE_Tasks function 289
SYS_CONSOLE_Write function 291
SYS_DEBUG macro 282
sys_debug.h 309
SYS_DEBUG_BreakPoint macro 305
SYS_DEBUG_ErrorLevelGet function 305
SYS_DEBUG_ErrorLevelSet function 306
SYS_DEBUG_INDEX_0 macro 307
SYS_DEBUG_Initialize function 301
SYS_DEBUG_Message function 303
SYS_DEBUG_MESSAGE macro 276
SYS_DEBUG_Print function 304
SYS_DEBUG_PRINT macro 277
SYS_DEBUG_Status function 302
SYS_DEBUG_Tasks function 303
sys_dma.h 115
SYS_ERROR macro 282
SYS_ERROR_LEVEL enumeration 307
SYS_ERROR_PRINT macro 278
sys_fs.h 405
sys_fs_config_template.h 407
SYS_FS_CurrentDriveGet function 366
SYS_FS_CurrentDriveSet function 367
SYS_FS_CurrentWorkingDirectoryGet function 356
SYS_FS_DirClose function 350
SYS_FS_DirectoryChange function 355
SYS_FS_DirectoryMake function 349
SYS_FS_DirOpen function 349
SYS_FS_DirRead function 351
SYS_FS_DirRewind function 353
SYS_FS_DirSearch function 354
SYS_FS_DriveFormat function 370
SYS_FS_DriveLabelGet function 367
SYS_FS_DriveLabelSet function 369
SYS_FS_DrivePartition function 371
SYS_FS_DriveSectorGet function 373
SYS_FS_ERROR enumeration 385
SYS_FS_Error function 361
SYS_FS_EVENT enumeration 395
SYS_FS_EVENT_HANDLER type 396
SYS_FS_EventHandlerSet function 374
SYS_FS_FILE_DIR_ATTR enumeration 393
SYS_FS_FILE_OPEN_ATTRIBUTES enumeration 390
SYS_FS_FILE_SEEK_CONTROL enumeration 386
SYS_FS_FILE_SYSTEM_TYPE enumeration 392
SYS_FS_FileCharacterPut function 333
SYS_FS_FileClose function 333
SYS_FS_FileDirectoryModeSet function 357
SYS_FS_FileDirectoryRemove function 358
SYS_FS_FileDirectoryRenameMove function 359
SYS_FS_FileDirectoryTimeSet function 360
SYS_FS_FileEOF function 334
SYS_FS_FileError function 335
SYS_FS_FileNameGet function 336
SYS_FS_FileOpen function 331
SYS_FS_FilePrintf function 337

SYS_FS_FileRead function 338
 SYS_FS_FileSeek function 339
 SYS_FS_FileSize function 340
 SYS_FS_FileStat function 341
 SYS_FS_FileStringGet function 342
 SYS_FS_FileStringPut function 343
 SYS_FSFileSync function 344
 SYS_FS_FileTell function 345
 SYS_FS_FileTestError function 346
 SYS_FS_FileTruncate function 347
 SYS_FS_FileWrite function 348
 SYS_FS_FORMAT enumeration 395
 SYS_FS_FSTAT structure 387
 SYS_FS_FUNCTIONS structure 387
 SYS_FS_HANDLE type 393
 SYS_FS_HANDLE_INVALID macro 393
 SYS_FS_Initialize function 362
 SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE type 402
 SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID macro 396
 SYS_FS_MEDIA_BLOCK_EVENT enumeration 397
 SYS_FS_MEDIA_COMMAND_STATUS enumeration 398
 SYS_FS_MEDIA_EVENT_HANDLER type 403
 SYS_FS_MEDIA_FUNCTIONS structure 398
 SYS_FS_MEDIA_GEOMETRY type 403
 SYS_FS_MEDIA_HANDLE type 403
 SYS_FS_MEDIA_HANDLE_INVALID macro 397
 sys_fs_media_manager.h 407
 SYS_FS_MEDIA_MANAGER_AddressGet function 381
 SYS_FS_MEDIA_MANAGER_CommandStatusGet function 383
 SYS_FS_MEDIA_MANAGER_DeRegister function 381
 SYS_FS_MEDIA_MANAGER_EventHandlerSet function 384
 SYS_FS_MEDIA_MANAGER_GetMediaGeometry function 384
 SYS_FS_MEDIA_MANAGER_MediaStatusGet function 382
 SYS_FS_MEDIA_MANAGER_Read function 377
 SYS_FS_MEDIA_MANAGER_Register function 379
 SYS_FS_MEDIA_MANAGER_RegisterTransferHandler function 380
 SYS_FS_MEDIA_MANAGER_SectorRead function 378
 SYS_FS_MEDIA_MANAGER_SectorWrite function 378
 SYS_FS_MEDIA_MANAGER_Tasks function 376
 SYS_FS_MEDIA_MANAGER_TransferTask function 376
 SYS_FS_MEDIA_MANAGER_VolumePropertyGet function 382
 SYS_FS_MEDIA_MOUNT_DATA structure 399
 SYS_FS_MEDIA_PROPERTY enumeration 400
 SYS_FS_MEDIA_REGION_GEOMETRY type 404
 SYS_FS_MEDIA_STATE enumeration 401
 SYS_FS_MEDIA_STATUS enumeration 401
 SYS_FS_MEDIA_TYPE enumeration 402
 SYS_FS_Mount function 364
 SYS_FS_REGISTRATION_TABLE structure 389
 SYS_FS_RESULT enumeration 390
 SYS_FS_Tasks function 363
 SYS_FS_TIME union 394
 SYS_FS_Unmount function 365
 SYS_FS_VOLUME_PROPERTY structure 404
 sys_int.h 423
 SYS_INT_Disable function 416
 SYS_INT_DynamicRegister@INT_SOURCE@SYS_INT_TASKS_POINT
 ER@SYS_MODULE_OBJ 422
 SYS_INT_Enable function 416
 SYS_INT_IsEnabled function 417
 SYS_INT_Restore function 417
 SYS_INT_SourceDisable function 418
 SYS_INT_SourceEnable function 419
 SYS_INT_SourceIsEnabled function 421
 SYS_INT_SourceStatusClear function 419
 SYS_INT_SourceStatusGet function 420
 SYS_INT_SourceStatusSet function 421
 SYS_MEDIA_BLOCK_COMMAND_HANDLE type 434
 SYS_MEDIA_BLOCK_EVENT enumeration 434
 SYS_MEDIA_COMMAND_STATUS enumeration 435
 SYS_MEDIA_EVENT_HANDLER type 435
 SYS_MEDIA_GEOMETRY structure 432
 SYS_MEDIA_GEOMETRY_TABLE_ERASE_ENTRY macro 432
 SYS_MEDIA_GEOMETRY_TABLE_READ_ENTRY macro 432
 SYS_MEDIA_GEOMETRY_TABLE_WRITE_ENTRY macro 432
 SYS_MEDIA_PROPERTY enumeration 433
 SYS_MEDIA_REGION_GEOMETRY structure 433
 SYS_MEDIA_STATUS enumeration 436
 SYS_MESSAGE macro 278
 SYS_MODULE_DEINITIALIZE_ROUTINE type 211
 SYS_MODULE_INDEX type 208
 SYS_MODULE_INIT union 113
 SYS_MODULE_INITIALIZE_ROUTINE type 212
 SYS_MODULE_OBJ type 208
 SYS_MODULE_OBJ_INVALID macro 209
 SYS_MODULE_OBJ_STATIC macro 215
 SYS_MODULE_REINITIALIZE_ROUTINE type 213
 SYS_MODULE_STATUS_ROUTINE type 214
 SYS_MODULE_TASKS_ROUTINE type 214
 sys_ports.h 424
 SYS_PRINT macro 278
 SYS_STATUS enumeration 209
 sys_time.h 467
 SYS_TIME_CALLBACK type 462
 SYS_TIME_CALLBACK_TYPE enumeration 463
 SYS_TIME_CallbackRegisterMS function 452
 SYS_TIME_CallbackRegisterUS function 453
 SYS_TIME_Counter64Get function 456
 SYS_TIME_CounterGet function 457
 SYS_TIME_CounterSet function 458
 SYS_TIME_CountToMS function 458
 SYS_TIME_CountToUS function 459
 sys_time_definitions.h 468
 SYS_TIME_Deinitialize function 445
 SYS_TIME_DelayIsComplete function 455
 SYS_TIME_DelayMS function 454
 SYS_TIME_DelayUS function 454
 SYS_TIME_FrequencyGet function 461
 SYS_TIME_HANDLE type 464
 SYS_TIME_HANDLE_INVALID macro 464
 SYS_TIME_INIT type 462
 SYS_TIME_Initialize function 443
 SYS_TIME_MSToCount function 460
 sys_time_multiclient 563
 SYS_TIME_PLIB_CALLBACK type 464

- SYS_TIME_PLIB_CALLBACK_REGISTER type 465
SYS_TIME_PLIB_COMPARE_SET type 465
SYS_TIME_PLIB_COUNTER_GET type 465
SYS_TIME_PLIB_FREQUENCY_GET type 465
SYS_TIME_PLIB_INTERFACE structure 465
SYS_TIME_PLIB_PERIOD_SET type 466
SYS_TIME_PLIB_START type 466
SYS_TIME_PLIB_STOP type 466
SYS_TIME_RESULT enumeration 463
SYS_TIME_Status function 444
SYS_TIME_TimerCounterGet function 449
SYS_TIME_TimerCreate function 446
SYS_TIME_TimerDestroy function 447
SYS_TIME_TimerPeriodHasExpired function 450
SYS_TIME_TimerReload function 451
SYS_TIME_TimerStart function 448
SYS_TIME_TimerStop function 448
SYS_TIME_USToCount function 460
SYS_VersionGet macro 431
SYS_VersionStrGet macro 431
System Access 95
System Configuration 104
System Configurations 16
System Service Libraries Help 259
System Services Applications 563
System Services Libraries Overview 424
System State Machine 425
system.h 115
system_common.h 437
system_config.h 16
system_definitions.h 21
system_exceptions.c 21
system_init.c 18
system_interrupt.c 20
system_media.h 437
system_module.h 438
system_tasks.c 20
- T**
- task_notification_freertos 561
The Application File(s) 14
The Configuration-specific "framework" Folder 22
The Main File 13
Time System Service Applications 563
Time System Service Library Help 439
Trademarks 567
Typographic Conventions 568
- U**
- UART Console Device 285
UART Console Device Configuration Options 286
USART Driver Applications 532
USART Driver Library Help 233
uart_echo 533, 537
uart_multi_instance 535, 539
Using a Driver in an Application 27
Using a Driver's Client Interface 26
Using a Driver's System Interface 23
- Using Asynchronous and Callback Functions 31
Using Driver Interface Functions 30
Using the File System 326
Using the Help 567
Using the Library 36, 56, 72, 94, 116, 157, 183, 217, 234, 259, 284, 298, 311, 323, 408, 424, 439, 471
 Command Processor System Service Library 259
 Console System Service Library 284
 Debug System Service Library 298
 DMA System Service Library 311
 Interrupt System Service Library 408
 OSAL Library 471
 Port System Service Library 424
 SPI Driver Library 183
 USART Driver Library 234
Using the SYS_ASSERT Macro 429
- W**
- Web Resources 3
What is MPLAB Harmony? 9
Where to Begin 2