LUMINARY MICRO®

Stellaris® Peripheral Driver Library

USER'S GUIDE

PDL-LM3S-UG-3416

Legal Disclaimers and Trademark Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH LUMINARY MICRO PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN LUMINARY MICRO'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, LUMINARY MICRO ASSUMES NO LIABILITY WHATSOEVER, AND LUMINARY MICRO DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF LUMINARY MICRO'S PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. LUMINARY MICRO'S PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE-SUSTAINING APPLICATIONS.

Luminary Micro may make changes to specifications and product descriptions at any time, without notice. Contact your local Luminary Micro sales office or your distributor to obtain the latest specifications and before placing your product order.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Luminary Micro reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Luminary Micro, Inc.
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

Revision Information

This is version 3416 of this document, last updated on September 29, 2008.

Table of Contents

1 Introduction

The Luminary Micro® Stellaris® Peripheral Driver Library is a set of drivers for accessing the peripherals found on the Stellaris family of ARM® Cortex™-M3 based microcontrollers. While they are not drivers in the pure operating system sense (that is, they do not have a common interface and do not connect into a global device driver infrastructure), they do provide a mechanism that makes it easy to use the device's peripherals.

The capabilities and organization of the drivers are governed by the following design goals:

- They are written entirely in C except where absolutely not possible.
- They demonstrate how to use the peripheral in its common mode of operation.
- They are easy to understand.
- They are reasonably efficient in terms of memory and processor usage.
- They are as self-contained as possible.
- Where possible, computations that can be performed at compile time are done there instead of at run time.
- They can be built with more than one tool chain.

Some consequences of these design goals are:

- The drivers are not necessarily as efficient as they could be (from a code size and/or execution speed point of view). While the most efficient piece of code for operating a peripheral would be written in assembly and custom tailored to the specific requirements of the application, further size optimizations of the drivers would make them more difficult to understand.
- The drivers do not support the full capabilities of the hardware. Some of the peripherals provide complex capabilities which can not be utilized by the drivers in this library, though the existing code can be used as a reference upon which to add support for the additional capabilities.
- The APIs have a means of removing all error checking code. Since the error checking is usually only useful during initial program development, it can be removed to improve code size and speed.

For many applications, the drivers can be used as is. But in some cases, the drivers will have to be enhanced or rewritten in order to meet the functionality, memory, or processing requirements of the application. If so, the existing driver can be used as a reference on how to operate the peripheral.

The following tool chains are supported:

- Keil™ RealView® Microcontroller Development Kit
- CodeSourcery Sourcery G++ for Stellaris EABI
- IAR Embedded Workbench®
- Code Red Technologies tools

Source Code Overview

The following is an overview of the organization of the peripheral driver library source code, along with references to where each portion is described in detail.

`EULA.txt`	The full text of the End User License Agreement that covers the use of this software package.
`Makefile`	The rules for building the peripheral driver library. The contents of this file are described in chapter 2.
`asmdefs.h`	A set of macros used by assembly language source files. The contents of this file are described in chapter 30.
`boards/`	This directory contains the source code for the example applications that run on the various Luminary Micro development and evaluation boards. These applications are described in chapter 31 through 49.
`boot_loader/`	This directory contains the source code for the boot loader. This code is described in chapter 29.
`codered/`	This directory contains the source files that are specific to the Code Red Technologies tool chain. The contents of this directory are described in chapters 3 and 30.
`ewarm/`	This directory contains the source files that are specific to the IAR Embedded Workbench tool chain. The contents of this directory are described in chapters 3 and 30.
`gcc/`	This directory contains the source files that are specific to the GNU tool chain. The contents of this directory are described in chapters 3 and 30.
`grlib/`	This directory contains the Stellaris Graphics Library. The contents of this directory are described in a PDF contained within this directory.
`hw_*.h`	Header files, one per peripheral, that describe all the registers and the bit fields within those registers for each peripheral. These header files are used by the drivers to directly access a peripheral, and can be used by application code to bypass the peripheral driver library API.
`inc/`	This directory holds the part specific header files used for the direct register access programming model, which is described in chapter 4.
`makedefs`	A set of definitions used by make files. The contents of this file are described in chapter 30.
`rvmdk/`	This directory contains the source files that are specific to the Keil RealView Microcontroller Development Kit. The contents of this directory are described in chapters 3 and 30.
`src/`	This directory contains the source code for the drivers, which are each described in chapters 5 through 25.
`third_party/`	This directory contains third party software packages that have been ported to the Stellaris microcontroller family. Each package has its own documentation describing its functionality.

`usblib/`	This directory contains the Stellaris USB Library. The contents of this directory are described in a PDF contained within this directory.
`utils/`	This directory contains a set of utility functions for use by the example applications. The contents of this directory are described in chapter 27.

2 Building The Code

2.1 Required Software

In order to build the code in the peripheral driver library, the following software is needed.

- One of the following tool chains:
  - Keil RealView Microcontroller Development Kit
  - CodeSourcery's Sourcery G++ for ARM EABI
  - IAR Embedded Workbench
  - Code Red Technologies tools
- If building from the command line, some form of Unix environment for Windows®.

Follow the directions provided with the tool chain of choice to install the compiler and debugger (there are also Quickstart guides provided by Luminary Micro that describe installing each of these tool chains); this will also add the compiler to the search path so that it can be executed.

Once the required software is installed, the peripheral driver library source must be extracted from its ZIP file using the archiver of your choice (such as WinZip® or the built-in utilities in Windows XP). For the remainder of these directions, it will be assumed that the source is extracted to `c:/DriverLib`.

2.2 Building With Keil uVision

The peripheral driver library and each example application has a uVision project (with a `.Uv2` file name extension) that can be used to build from within uVision. Simply load the project file into uVision and click on the "Build target" or "Rebuild all target files" buttons. Note that the peripheral driver library (`c:/DriverLib/src/driverlib.Uv2`) project must be built before any of the example applications can be built.

There is a multi-project workspace file (with a `.mpw` file name extension) that includes all the projects for a particular board in each board directory. For example, in the `boards/dk-lm3s101` directory, there is a `dk-lm3s101.mpw` file that contains the project for the peripheral driver library along with the projects for all of the board examples for the DK-LM3S101 board.

See the "RealView Quickstart" for details about using uVision.

2.3 Building with IAR Embedded Workbench

The peripheral driver library and each example application has an Embedded Workbench project (with a `.ewp` file name extension) that can be used to build from within Embedded Workbench version 5. Simply load the project file into Embedded Workbench and select "Make" or "Rebuild All" from the "Project" menu. Note that the peripheral driver library (`c:/DriverLib/src/driverlib.ewp`) project must be built before any of the examples applications can be built.

There is a workspace file (with a `.eww` file name extension) that includes all the projects for a particular board in each board directory. For example, in the `boards/dk-lm3s101` directory, there is a `dk-lm3s101.eww` file that contains the project for the peripheral driver library along with the projects for all of the board examples for the DK-LM3S101 board.

There are also versions of these files for use with Embedded Workbench version 4.42a. They are `*-ewarm4.ewp` and `*-ewarm4.eww`, and are located in the same places as the version 5 files.

See the "IAR KickStart Quickstart" for details about using Embedded Workbench.

2.4 Building with CodeSourcery Sourcery G++

The peripheral driver library and each example application can be built utilizing the CodeSourcery Common Startup Code Sequence (CS3). By setting the **COMPILER** environment variable to "`sourcerygxx`", CS3 will be used to build the application. The advantage of using CS3 is the ability to easily use the C library provided by CodeSourcery for things such as printf().

See the CodeSourcery Getting Started document for details on CS3 and how to use it in your application. See the following section for details on how to build the code using CS3.

2.5 Building with Code Red Technologies Tools

The peripheral driver library and each example application can be built using the compiler supplied with Code Red Technologies tools. By setting the **COMPILER** environment variable to "`codered`", the Code Red Technologies tools will be used when building from the command line or from within the Code Red Technologies development environment.

See the "code_red Quickstart" for details about using Code Red Technologies tools.

2.6 Building From The Command Line

In order to build from the command line, some form of Unix environment for Windows is required. The recommended solution is the Unix Utilities from SourceForge (`http://unxutils.sourceforge.net`); alternatives are Cygwin (`http://www.cygwin.com`) and MinGW (`http://www.mingw.org`). The Unix Utilities and Cygwin have both been tested and work with this library; MinGW should work as well, though it has not been tested.

See the "GNU Quickstart" for details about installing and setting up the Unix Utilities.

The makefiles will not work with the `make` utilities normally available on Windows (such as the one provided with RealView); the "Unix" version must appear in the search path before any other version of `make`. Of course, if using a compiler on Linux, the normal Posix shell environment that exists is more than adequate for building the code.

The Unix utilities from SourceForge are in a ZIP file that must be unpacked; for the remainder of these directions, it will be assumed that the Unix utilities are extracted to `c:/`.

The search path must be manually updated to include both the `c:/bin` directory and the `c:/usr/local/wbin` directory, preferably at the beginning of the search path (so that `make` from `c:/usr/local/wbin` is used in preference to other versions of `make`).

The remainder of these directions assume that the shell in `c:/bin/sh` is being used in preference to the command shell provided by Windows XP; if not using this shell, the commands may have to be modified to be compatible with the Windows XP shell.

A couple of quick tests will determine if the search path is set up correctly. First, type:

```
make --version
```

It should report back that some version of GNU Make was invoked; if not, then the wrong `make` utility is being found and the search path needs to be corrected. Next, type:

```
type sh
```

It should specify the path where `sh.exe` from the Unix utilities was extracted; if not, then the make utility will not be able to find the shell (meaning that the build will fail) and the search path needs to be corrected.

If using the Keil RealView Microcontroller Development Kit, the following will verify that the compiler can be found (which means that all the other tool chain utilities will be found as well):

```
type armcc
```

If using CodeSourcery's Sourcery G++ for ARM EABI, the following will verify that the compiler can be found:

```
type arm-stellaris-eabi-gcc
```

If using IAR Embedded Workbench, the following will verify that the compiler can be found:

```
type iccarm
type xlink
```

If using Code Red Technologies tools, the following will verify that the compiler can be found:

```
type arm-none-eabi-gcc
```

If any of the above checks fail then the build will probably fail as well. In each case, the search path would need to be updated so that the tools in question can be located by the shell.

Now, to build the library and example applications, type the following:

```
cd c:/DriverLib
make
```

It will display short messages to indicate the build step being performed; the following extract is an example:

```
...
CC    timer.c
CC    uart.c
CC    watchdog.c
AR    gcc/libdriver.a
...
```

This indicates that it is compiling `timer.c`, `uart.c`, `watchdog.c`, and then creating a library called `gcc/libdriver.a`. Displaying short messages like this makes it very easy to spot warnings and errors encountered during the build process.

There are several variables that control the build process. They can either appear as environment variables or they can be passed on the command line to make. The variables are:

- **COMPILER** specifies the tool chain to be used to build the source code. Currently, this can be `codered`, `ewarm`, `gcc`, `rvmdk`, or `sourcerygxx`; the default value if not specified is `gcc`.
- **DEBUGGER** specifies the debugger that will be used to run the executables. This affects the version of the Diag...() functions that are used. Currently, this can be either `cspy`, `gdb`, or `uvision`; the default value if not specified depends on the value of **COMPILER** (`codered` results in `gdb`, `ewarm` results in `cspy`, `gcc` results in `gdb`, `rvmdk` results in `uvision`, and `sourcerygxx` results in `gdb`).
- **DEBUG** specifies that debugging information should be included in the object files that are built. This allows the debugger to perform source level debugging, and may add additional code that helps the development and debugging process (such as the ASSERT-based error checking). The value of this variable is not important; if it exists then debugging information will be included. If not specified, debugging information is not included.
- **VERBOSE** specifies that the actual compiler invocations should be displayed instead of the brief build steps. The value of this variable is not important; if it exists then verbose mode will be enabled. If not specified, verbose mode is disabled.

So, for example, to build using `rvmdk` with debugging enabled, type the following:

```
make COMPILER=rvmdk DEBUG=1
```

Alternatively, the following could be typed instead:

```
export COMPILER=rvmdk
export DEBUG=1
make
```

The advantage of the latter is subsequent builds require only invoking `make`, and is less prone to unexpected results from forgetting to add the variables to the command line each time (that is, from mixing and matching objects built with different definitions).

To remove all build products, use:

```
make clean
```

Note that this still depends upon the **COMPILER** environment variable; it will only remove the objects associated with the tool chain in use (that is, it can be used to clean out the `rvmdk` objects while leaving the `gcc` objects undisturbed).

3 Boot Code

The boot code contains the minimal set of code required to set up the vector table and get application code running after a system reset. There are multiple versions of the boot code, one per supported tool chain (some tool chain-specific constructs are used to find where the code, data, and bss segments reside in memory); the startup code is contained in `<toolchain>/startup.c`. Accompanying the startup code is the corresponding linker script that is used to link an application so that the vector table, code segment, data segment initializers, and data segments are placed in the appropriate locations in memory; this script is contained in `<toolchain>/standalone.ld` (`standalone.xcl` for IAR Embedded Workbench).

The boot code and its corresponding linker script utilize a typical memory layout for a flash-based system. The first portion of the flash is used for code and read-only data (this is referred to as the "code" segment). Immediately following are the initializers (if any) for the non-zero initialized data. The first portion of SRAM is used for the non-zero initialized data (referred to as the "data" segment), with the zero initialized data immediately following it (referred to as the "bss" segment).

The vector table of the Cortex-M3 microprocessor contains four required entries; they are the initial stack pointer, the reset handler address, the NMI handler address, and the hard fault handler address. Upon reset, the processor will load the initial stack pointer and then start executing the reset handler. The initial stack pointer is required since an NMI or hard fault can occur at any time; the stack is required to be able to take those interrupts since the processor will automatically push eight items onto the stack.

The `g_pfnVectors` array contains a full vector table. It contains the addresses of the all the handlers and the end of the initial stack. Tool chain-specific constructs are used to provide a hint to the linker that is used to make sure that this array is located at `0x0000.0000`, the default location of the vector table.

The `NmiSR` function contains the NMI handler. It simply enters an infinite loop, effectively halting the application if an NMI occurs; the application state is therefore preserved for examination by a debugger. If desired, the application can provide its own NMI handler via the interrupt driver.

The `FaultISR` function contains the hard fault handler. It also enters an infinite loop and can be replaced by the application.

The `ResetISR` function contains the reset handler. It copies the initializers from the end of the code segment in flash into the data segment in SRAM, zero fills the bss segment, and branches to the application-supplied entry point. This corresponds to the minimal set of things that must be done for C code to work properly when called; anything more complicated that is required by an application must be provided by that application.

The application must supply an entry point called `main` that takes no arguments and never returns. This function will be called by `ResetISR` after memory has been initialized. If `main` does return, `ResetISR` will also return, which will cause a hard fault to occur.

Each example application has its own copy of the boot code with the required interrupt handlers in place. This allows the interrupt handlers to be custom to each example and reside in flash.

4 Programming Model

4.1 Introduction

The peripheral driver library provides support for two programming models: the direct register access model and the software driver model. Each model can be used independently, or combined, based on the needs of the application or the programming environment desired by the developer.

Each programming model has advantages and disadvantages. Use of the direct register access model will generally result in smaller and more efficient code than using the software driver model. However, the direct register access model does require detailed knowledge of the operation of each register, bit field, their interactions, and any sequencing required for proper operation of the peripheral; the developer is insulated from these details by the software driver model, generally requiring less time to develop applications.

4.2 Direct Register Access Model

In the direct register access model, the peripherals are programmed by the application by writing values directly into the peripheral's registers. A set of macros is provided that simplifies this process. These macros are stored in part-specific header files contained in the `inc` directory; the name of the header file matches the part number (for example, the header file for the LM3S6965 microcontroller is `inc/lm3s6965.h`). By including the header file that matches the part being used, macros are available for accessing all registers on that part, as well as all bit fields within those registers. No macros are available for registers that do not exist on the part in question, making it difficult to access registers that do not exist.

The defines used by the direct register access model follow a naming convention that makes it easier to know how to use a particular macro. The rules are as follows:

- Values that end in `_R` are used to access the value of a register. For example, `SSI0_CR0_R` is used to access the `CR0` register in the SSI0 module.
- Values that end in `_M` represent the mask for a multi-bit field in a register. If the value placed in the multi-bit field is a number, there will be a macro with the same base name but ending with `_S` (for example, `SSI_CR0_SCR_M` and `SSI_CR0_SCR_S`). If the value placed into the multi-bit field is an enumeration, then there will be a set of macros with the same base name but ending with identifiers for the various enumeration values (for example, the `SSI_CR0_FRF_M` macro defines the bit field, and the `SSI_CR0_FRF_NMW`, `SSI_CR0_FRF_TI`, and `SSI_CR0_FRF_MOTO` macros provide the enumerations for the bit field).
- Values that end in `_S` represent the number of bits to shift a value in order to align it with a multi-bit field. These values will match the macro with the same base name but ending with `_M`.

- All other macros represent the value of a bit field.
- All register name macros start with the module name and instance number (for example, SSI0 for the first SSI module) and are followed by the name of the register as it appears in the data sheet (for example, the CR0 register in the data sheet results in SSI0_CR0_R).
- All register bit fields start with the module name, followed by the register name, and then followed by the bit field name as it appears in the data sheet. For example, the SCR bit field in the CR0 register in the SSI module will be identified by SSI_CR0_SCR.... In the case where the bit field is a single bit, there will be nothing further (for example, SSI_CR0_SPH is a single bit in the CR0 register). If the bit field is more than a single bit, there will be a mask value (_M) and either a shift (_S) if the bit field contains a number or a set of enumerations if not.

Given these definitions, the CR0 register can be programmed as follows:

```
SSI0_CR0_R = ((5 << SSI_CR0_SCR_S) | SSI_CR0_SPH | SSI_CR0_SPO |
              SSI_CR0_FRF_MOTO | SSI_CR0_DSS_8);
```

Alternatively, the following has the same effect (although it is not as easy to understand):

```
SSI0_CR0_R = 0x000005c7;
```

Extracting the value of the SCR field from the CR0 register is as follows:

```
ulValue = (SSI0_CR0_R & SSI_CR0_SCR_M) >> SSI0_CR0_SCR_S;
```

The GPIO modules have many registers that do not have bit field definitions. For these registers, the register bits represent the individual GPIO pins; so bit zero in these registers corresponds to the **Px0** pin on the part (where **x** is replaced by a GPIO module letter), bit one corresponds to the **Px1** pin, and so on.

The blinky example for each board utilizes the direct register access model to blink the on-board LED.

Note:
The hw_*.h header files that are used by the drivers in the library contain many of the same definitions as the header files used for direct register access. As such, the two can not be included into the same source file without the compiler producing warnings about the redefinition of symbols.

4.3 Software Driver Model

In the software driver model, the API provided by the peripheral driver library is used by applications to control the peripherals. Since these drivers provide complete control of the peripherals in their normal mode of operation, it is possible to write an entire application without direct access to the hardware. This provides for rapid development of the application without requiring detailed knowledge of how to program the peripherals.

Corresponding to the direct register access model example, the following will also program the CR0 register in the SSI module (though that fact is hidden by the API):

```
SSIConfigSetExpClk(SSI0_BASE, 50000000, SSI_FRF_MOTO_MODE_3,
                   SSI_MODE_MASTER, 1000000, 8);
```

The resulting value in the `CR0` register might not be exactly the same since SSIConfigSetExpClk() may compute a different value for the `SCR` bit field than what was used in the direct register access model example.

All example applications other than `blinky` utilize the software driver model.

The drivers in the peripheral driver library are described in chapters 5 through 25 . They combine to form the software driver model.

4.4 Combining The Models

The direct register access model and software driver model can be utilized together in a single application. This allows the most appropriate model to be used in any particular situation within the application; for example, the software driver model can be used to configure the peripherals (since this is not performance critical) and the direct register access model can be used for operation of the peripheral (which may be more performance critical). Or, the software driver model can be used for peripherals that are not performance critical (such as a UART used for data logging) and the direct register access model for performance critical peripherals (such as the ADC module used to capture real-time analog data).

5 Analog Comparator

5.1 Introduction

The comparator API provides a set of functions for dealing with the analog comparators. A comparator can compare a test voltage against individual external reference voltage, a shared single external reference voltage, or a shared internal reference voltage. It can provide its output to a device pin, acting as a replacement for an analog comparator on the board, or it can be used to signal the application via interrupts or triggers to the ADC to cause it to start capturing a sample sequence. The interrupt generation and ADC triggering logic is separate, so that an interrupt can be generated on a rising edge and the ADC triggered on a falling edge (for example).

This driver is contained in `src/comp.c`, with `src/comp.h` containing the API definitions for use by applications.

5.2 API Functions

Functions

- void ComparatorConfigure (unsigned long ulBase, unsigned long ulComp, unsigned long ulConfig)
- void ComparatorIntClear (unsigned long ulBase, unsigned long ulComp)
- void ComparatorIntDisable (unsigned long ulBase, unsigned long ulComp)
- void ComparatorIntEnable (unsigned long ulBase, unsigned long ulComp)
- void ComparatorIntRegister (unsigned long ulBase, unsigned long ulComp, void (*pfnHandler)(void))
- tBoolean ComparatorIntStatus (unsigned long ulBase, unsigned long ulComp, tBoolean bMasked)
- void ComparatorIntUnregister (unsigned long ulBase, unsigned long ulComp)
- void ComparatorRefSet (unsigned long ulBase, unsigned long ulRef)
- tBoolean ComparatorValueGet (unsigned long ulBase, unsigned long ulComp)

5.2.1 Detailed Description

The comparator API is fairly simple, like the comparators themselves. There are functions for configuring a comparator and reading its output (ComparatorConfigure(), ComparatorRefSet() and ComparatorValueGet()) and functions for dealing with an interrupt handler for the comparator (ComparatorIntRegister(), ComparatorIntUnregister(), ComparatorIntEnable(), ComparatorIntDisable(), ComparatorIntStatus(), and ComparatorIntClear()).

5.2.2 Function Documentation

5.2.2.1 ComparatorConfigure

Configures a comparator.

Prototype:

```
void
ComparatorConfigure(unsigned long ulBase,
                    unsigned long ulComp,
                    unsigned long ulConfig)
```

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator to configure.

ulConfig is the configuration of the comparator.

Description:

This function will configure a comparator. The *ulConfig* parameter is the result of a logical OR operation between the **COMP_TRIG_xxx**, **COMP_INT_xxx**, **COMP_ASRCP_xxx**, and **COMP_OUTPUT_xxx** values.

The **COMP_TRIG_xxx** term can take on the following values:

- **COMP_TRIG_NONE** to have no trigger to the ADC.
- **COMP_TRIG_HIGH** to trigger the ADC when the comparator output is high.
- **COMP_TRIG_LOW** to trigger the ADC when the comparator output is low.
- **COMP_TRIG_FALL** to trigger the ADC when the comparator output goes low.
- **COMP_TRIG_RISE** to trigger the ADC when the comparator output goes high.
- **COMP_TRIG_BOTH** to trigger the ADC when the comparator output goes low or high.

The **COMP_INT_xxx** term can take on the following values:

- **COMP_INT_HIGH** to generate an interrupt when the comparator output is high.
- **COMP_INT_LOW** to generate an interrupt when the comparator output is low.
- **COMP_INT_FALL** to generate an interrupt when the comparator output goes low.
- **COMP_INT_RISE** to generate an interrupt when the comparator output goes high.
- **COMP_INT_BOTH** to generate an interrupt when the comparator output goes low or high.

The **COMP_ASRCP_xxx** term can take on the following values:

- **COMP_ASRCP_PIN** to use the dedicated Comp+ pin as the reference voltage.
- **COMP_ASRCP_PIN0** to use the Comp0+ pin as the reference voltage (this the same as **COMP_ASRCP_PIN** for the comparator 0).
- **COMP_ASRCP_REF** to use the internally generated voltage as the reference voltage.

The **COMP_OUTPUT_xxx** term can take on the following values:

- **COMP_OUTPUT_NORMAL** to enable a non-inverted output from the comparator to a device pin.
- **COMP_OUTPUT_INVERT** to enable an inverted output from the comparator to a device pin.

- **COMP_OUTPUT_NONE** is deprecated and behaves the same as **COMP_OUTPUT_NORMAL**.

Returns:
None.

5.2.2.2 ComparatorIntClear

Clears a comparator interrupt.

Prototype:
```
void
ComparatorIntClear(unsigned long ulBase,
                   unsigned long ulComp)
```

Parameters:
ulBase is the base address of the comparator module.
ulComp is the index of the comparator.

Description:
The comparator interrupt is cleared, so that it no longer asserts. This must be done in the interrupt handler to keep it from being called again immediately upon exit. Note that for a level triggered interrupt, the interrupt cannot be cleared until it stops asserting.

Note:
Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:
None.

5.2.2.3 ComparatorIntDisable

Disables the comparator interrupt.

Prototype:
```
void
ComparatorIntDisable(unsigned long ulBase,
                     unsigned long ulComp)
```

Parameters:
ulBase is the base address of the comparator module.
ulComp is the index of the comparator.

Description:
This function disables generation of an interrupt from the specified comparator. Only comparators whose interrupts are enabled can be reflected to the processor.

Returns:
None.

5.2.2.4 ComparatorIntEnable

Enables the comparator interrupt.

Prototype:
```
void
ComparatorIntEnable(unsigned long ulBase,
                    unsigned long ulComp)
```

Parameters:
ulBase is the base address of the comparator module.
ulComp is the index of the comparator.

Description:
This function enables generation of an interrupt from the specified comparator. Only comparators whose interrupts are enabled can be reflected to the processor.

Returns:
None.

5.2.2.5 ComparatorIntRegister

Registers an interrupt handler for the comparator interrupt.

Prototype:
```
void
ComparatorIntRegister(unsigned long ulBase,
                      unsigned long ulComp,
                      void (*pfnHandler)(void))
```

Parameters:
ulBase is the base address of the comparator module.
ulComp is the index of the comparator.
pfnHandler is a pointer to the function to be called when the comparator interrupt occurs.

Description:
This sets the handler to be called when the comparator interrupt occurs. This will enable the interrupt in the interrupt controller; it is the interrupt-handler's responsibility to clear the interrupt source via ComparatorIntClear().

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

5.2.2.6 ComparatorIntStatus

Gets the current interrupt status.

Prototype:

```
tBoolean
ComparatorIntStatus(unsigned long ulBase,
                    unsigned long ulComp,
                    tBoolean bMasked)
```

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator.

bMasked is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

Description:

This returns the interrupt status for the comparator. Either the raw or the masked interrupt status can be returned.

Returns:

true if the interrupt is asserted and **false** if it is not asserted.

5.2.2.7 ComparatorIntUnregister

Unregisters an interrupt handler for a comparator interrupt.

Prototype:

```
void
ComparatorIntUnregister(unsigned long ulBase,
                        unsigned long ulComp)
```

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator.

Description:

This function will clear the handler to be called when a comparator interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

5.2.2.8 ComparatorRefSet

Sets the internal reference voltage.

Prototype:

```
void
ComparatorRefSet(unsigned long ulBase,
                 unsigned long ulRef)
```

Parameters:

ulBase is the base address of the comparator module.

ulRef is the desired reference voltage.

Description:

This function will set the internal reference voltage value. The voltage is specified as one of the following values:

- **COMP_REF_OFF** to turn off the reference voltage
- **COMP_REF_0V** to set the reference voltage to 0 V
- **COMP_REF_0_1375V** to set the reference voltage to 0.1375 V
- **COMP_REF_0_275V** to set the reference voltage to 0.275 V
- **COMP_REF_0_4125V** to set the reference voltage to 0.4125 V
- **COMP_REF_0_55V** to set the reference voltage to 0.55 V
- **COMP_REF_0_6875V** to set the reference voltage to 0.6875 V
- **COMP_REF_0_825V** to set the reference voltage to 0.825 V
- **COMP_REF_0_928125V** to set the reference voltage to 0.928125 V
- **COMP_REF_0_9625V** to set the reference voltage to 0.9625 V
- **COMP_REF_1_03125V** to set the reference voltage to 1.03125 V
- **COMP_REF_1_134375V** to set the reference voltage to 1.134375 V
- **COMP_REF_1_1V** to set the reference voltage to 1.1 V
- **COMP_REF_1_2375V** to set the reference voltage to 1.2375 V
- **COMP_REF_1_340625V** to set the reference voltage to 1.340625 V
- **COMP_REF_1_375V** to set the reference voltage to 1.375 V
- **COMP_REF_1_44375V** to set the reference voltage to 1.44375 V
- **COMP_REF_1_5125V** to set the reference voltage to 1.5125 V
- **COMP_REF_1_546875V** to set the reference voltage to 1.546875 V
- **COMP_REF_1_65V** to set the reference voltage to 1.65 V
- **COMP_REF_1_753125V** to set the reference voltage to 1.753125 V
- **COMP_REF_1_7875V** to set the reference voltage to 1.7875 V
- **COMP_REF_1_85625V** to set the reference voltage to 1.85625 V
- **COMP_REF_1_925V** to set the reference voltage to 1.925 V
- **COMP_REF_1_959375V** to set the reference voltage to 1.959375 V
- **COMP_REF_2_0625V** to set the reference voltage to 2.0625 V
- **COMP_REF_2_165625V** to set the reference voltage to 2.165625 V
- **COMP_REF_2_26875V** to set the reference voltage to 2.26875 V
- **COMP_REF_2_371875V** to set the reference voltage to 2.371875 V

Returns:

None.

5.2.2.9 ComparatorValueGet

Gets the current comparator output value.

Prototype:

```
tBoolean
ComparatorValueGet(unsigned long ulBase,
                   unsigned long ulComp)
```

Parameters:

ulBase is the base address of the comparator module.

ulComp is the index of the comparator.

Description:

This function retrieves the current value of the comparator output.

Returns:

Returns **true** if the comparator output is high and **false** if the comparator output is low.

5.3 Programming Example

The following example shows how to use the comparator API to configure the comparator and read its value.

```
//
// Configure the internal voltage reference.
//
ComparatorRefSet(COMP_BASE, COMP_REF_1_65V);

//
// Configure a comparator.
//
ComparatorConfigure(COMP_BASE, 0,
                    (COMP_TRIG_NONE | COMP_INT_BOTH |
                     COMP_ASRCP_REF | COMP_OUTPUT_NONE));

//
// Delay for some time...
//

//
// Read the comparator output value.
//
ComparatorValueGet(COMP_BASE, 0);
```

6 Analog to Digital Converter (ADC)

6.1 Introduction

The analog to digital converter (ADC) API provides a set of functions for dealing with the ADC. Functions are provided to configure the sample sequencers, read the captured data, register a sample sequence interrupt handler, and handle interrupt masking/clearing.

The ADC supports up to eight input channels plus an internal temperature sensor. Four sampling sequences, each with configurable trigger events, can be captured. The first sequence will capture up to eight samples, the second and third sequences will capture up to four samples, and the fourth sequence will capture a single sample. Each sample can be the same channel, different channels, or any combination in any order.

The sample sequences have configurable priorities that determine the order in which they are captured when multiple triggers occur simultaneously. The highest priority sequence that is currently triggered will be sampled. Care must be taken with triggers that occur frequently (such as the "always" trigger); if their priority is too high it is possible to starve the lower priority sequences.

Beginning with Rev C0 of the Stellaris microcontroller, hardware oversampling of the ADC data is available for improved accuracy. An oversampling factor of 2x, 4x, 8x, 16x, 32x, and 64x is supported, but reduces the throughput of the ADC by a corresponding factor. Hardware oversampling is applied uniformly across all sample sequences.

Software oversampling of the ADC data is also available (even when hardware oversampling is available). An oversampling factor of 2x, 4x, and 8x is supported, but reduces the depth of the sample sequences by a corresponding amount. For example, the first sample sequence will capture eight samples; in 4x oversampling mode it can only capture two samples since the first four samples are used over the first oversampled value and the second four samples are used for the second oversampled value. The amount of software oversampling is configured on a per sample sequence basis.

A more sophisticated software oversampling can be used to eliminate the reduction of the sample sequence depth. By increasing the ADC trigger rate by 4x (for example) and averaging four triggers worth of data, 4x oversampling is achieved without any loss of sample sequence capability. In this case, an increase in the number of ADC triggers (and presumably ADC interrupts) is the consequence. Since this requires adjustments outside of the ADC driver itself, this is not directly supported by the driver (though nothing in the driver prevents it). The software oversampling APIs should not be used in this case.

This driver is contained in `src/adc.c`, with `src/adc.h` containing the API definitions for use by applications.

6.2 API Functions

Functions

- void ADCHardwareOversampleConfigure (unsigned long ulBase, unsigned long ulFactor)
- void ADCIntClear (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCIntDisable (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCIntEnable (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCIntRegister (unsigned long ulBase, unsigned long ulSequenceNum, void (*pfnHandler)(void))
- unsigned long ADCIntStatus (unsigned long ulBase, unsigned long ulSequenceNum, tBoolean bMasked)
- void ADCIntUnregister (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCProcessorTrigger (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCSequenceConfigure (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulTrigger, unsigned long ulPriority)
- long ADCSequenceDataGet (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long *pulBuffer)
- void ADCSequenceDisable (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCSequenceEnable (unsigned long ulBase, unsigned long ulSequenceNum)
- long ADCSequenceOverflow (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCSequenceOverflowClear (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCSequenceStepConfigure (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulStep, unsigned long ulConfig)
- long ADCSequenceUnderflow (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCSequenceUnderflowClear (unsigned long ulBase, unsigned long ulSequenceNum)
- void ADCSoftwareOversampleConfigure (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulFactor)
- void ADCSoftwareOversampleDataGet (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long *pulBuffer, unsigned long ulCount)
- void ADCSoftwareOversampleStepConfigure (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulStep, unsigned long ulConfig)

6.2.1 Detailed Description

The analog to digital converter API is broken into three groups of functions: those that deal with the sample sequences, those that deal with the processor trigger, and those that deal with interrupt handling.

The sample sequences are configured with ADCSequenceConfigure() and ADCSequenceStepConfigure(). They are enabled and disabled with ADCSequenceEnable() and ADCSequenceDisable(). The captured data is obtained with ADCSequenceDataGet(). Sample sequence FIFO overflow and underflow is managed with ADCSequenceOverflow(), ADCSequenceOverflowClear(), ADCSequenceUnderflow(), and ADCSequenceUnderflowClear().

Hardware oversampling of the ADC is controlled with ADCHardwareOversampleConfigure(). Software oversampling of the ADC is controlled with ADCSoftwareOversampleConfigure(), ADCSoftwareOversampleStepConfigure(), and ADCSoftwareOversampleDataGet().

The processor trigger is generated with ADCProcessorTrigger().

The interrupt handler for the ADC sample sequence interrupts are managed with ADCIntRegister() and ADCIntUnregister(). The sample sequence interrupt sources are managed with ADCIntDisable(), ADCIntEnable(), ADCIntStatus(), and ADCIntClear().

6.2.2 Function Documentation

6.2.2.1 ADCHardwareOversampleConfigure

Configures the hardware oversampling factor of the ADC.

Prototype:

```
void
ADCHardwareOversampleConfigure(unsigned long ulBase,
                               unsigned long ulFactor)
```

Parameters:

ulBase is the base address of the ADC module.

ulFactor is the number of samples to be averaged.

Description:

This function configures the hardware oversampling for the ADC, which can be used to provide better resolution on the sampled data. Oversampling is accomplished by averaging multiple samples from the same analog input. Six different oversampling rates are supported; 2x, 4x, 8x, 16x, 32x, and 64x. Specifying an oversampling factor of zero will disable hardware oversampling.

Hardware oversampling applies uniformly to all sample sequencers. It does not reduce the depth of the sample sequencers like the software oversampling APIs; each sample written into the sample sequence FIFO is a fully oversampled analog input reading.

Enabling hardware averaging increases the precision of the ADC at the cost of throughput. For example, enabling 4x oversampling reduces the throughput of a 250 Ksps ADC to 62.5 Ksps.

Note:

Hardware oversampling is available beginning with Rev C0 of the Stellaris microcontroller.

Returns:

None.

6.2.2.2 ADCIntClear

Clears sample sequence interrupt source.

Prototype:

```
void
ADCIntClear(unsigned long ulBase,
            unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

Description:
The specified sample sequence interrupt is cleared, so that it no longer asserts. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

Note:
Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:
None.

6.2.2.3 ADCIntDisable

Disables a sample sequence interrupt.

Prototype:
```
void
ADCIntDisable(unsigned long ulBase,
              unsigned long ulSequenceNum)
```

Parameters:
ulBase is the base address of the ADC module.
ulSequenceNum is the sample sequence number.

Description:
This function disables the requested sample sequence interrupt.

Returns:
None.

6.2.2.4 ADCIntEnable

Enables a sample sequence interrupt.

Prototype:
```
void
ADCIntEnable(unsigned long ulBase,
             unsigned long ulSequenceNum)
```

Parameters:
ulBase is the base address of the ADC module.
ulSequenceNum is the sample sequence number.

Description:
This function enables the requested sample sequence interrupt. Any outstanding interrupts are cleared before enabling the sample sequence interrupt.

Returns:
None.

6.2.2.5 ADCIntRegister

Registers an interrupt handler for an ADC interrupt.

Prototype:
```
void
ADCIntRegister(unsigned long ulBase,
               unsigned long ulSequenceNum,
               void (*pfnHandler)(void))
```

Parameters:
ulBase is the base address of the ADC module.
ulSequenceNum is the sample sequence number.
pfnHandler is a pointer to the function to be called when the ADC sample sequence interrupt occurs.

Description:
This function sets the handler to be called when a sample sequence interrupt occurs. This will enable the global interrupt in the interrupt controller; the sequence interrupt must be enabled with ADCIntEnable(). It is the interrupt handler's responsibility to clear the interrupt source via ADCIntClear().

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

6.2.2.6 ADCIntStatus

Gets the current interrupt status.

Prototype:
```
unsigned long
ADCIntStatus(unsigned long ulBase,
             unsigned long ulSequenceNum,
             tBoolean bMasked)
```

Parameters:
ulBase is the base address of the ADC module.
ulSequenceNum is the sample sequence number.
bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:
This returns the interrupt status for the specified sample sequence. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:
The current raw or masked interrupt status.

6.2.2.7 ADCIntUnregister

Unregisters the interrupt handler for an ADC interrupt.

Prototype:
```
void
ADCIntUnregister(unsigned long ulBase,
                 unsigned long ulSequenceNum)
```

Parameters:
ulBase is the base address of the ADC module.
ulSequenceNum is the sample sequence number.

Description:
This function unregisters the interrupt handler. This will disable the global interrupt in the interrupt controller; the sequence interrupt must be disabled via ADCIntDisable().

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

6.2.2.8 ADCProcessorTrigger

Causes a processor trigger for a sample sequence.

Prototype:
```
void
ADCProcessorTrigger(unsigned long ulBase,
                    unsigned long ulSequenceNum)
```

Parameters:
ulBase is the base address of the ADC module.
ulSequenceNum is the sample sequence number.

Description:
This function triggers a processor-initiated sample sequence if the sample sequence trigger is configured to **ADC_TRIGGER_PROCESSOR**.

Returns:
None.

6.2.2.9 ADCSequenceConfigure

Configures the trigger source and priority of a sample sequence.

Prototype:

```
void
ADCSequenceConfigure(unsigned long ulBase,
                     unsigned long ulSequenceNum,
                     unsigned long ulTrigger,
                     unsigned long ulPriority)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

ulTrigger is the trigger source that initiates the sample sequence; must be one of the **ADC_TRIGGER_*** values.

ulPriority is the relative priority of the sample sequence with respect to the other sample sequences.

Description:

This function configures the initiation criteria for a sample sequence. Valid sample sequences range from zero to three; sequence zero will capture up to eight samples, sequences one and two will capture up to four samples, and sequence three will capture a single sample. The trigger condition and priority (with respect to other sample sequence execution) is set.

The *ulTrigger* parameter can take on the following values:

- **ADC_TRIGGER_PROCESSOR** - A trigger generated by the processor, via the ADCProcessorTrigger() function.
- **ADC_TRIGGER_COMP0** - A trigger generated by the first analog comparator; configured with ComparatorConfigure().
- **ADC_TRIGGER_COMP1** - A trigger generated by the second analog comparator; configured with ComparatorConfigure().
- **ADC_TRIGGER_COMP2** - A trigger generated by the third analog comparator; configured with ComparatorConfigure().
- **ADC_TRIGGER_EXTERNAL** - A trigger generated by an input from the Port B4 pin.
- **ADC_TRIGGER_TIMER** - A trigger generated by a timer; configured with TimerControlTrigger().
- **ADC_TRIGGER_PWM0** - A trigger generated by the first PWM generator; configured with PWMGenIntTrigEnable().
- **ADC_TRIGGER_PWM1** - A trigger generated by the second PWM generator; configured with PWMGenIntTrigEnable().
- **ADC_TRIGGER_PWM2** - A trigger generated by the third PWM generator; configured with PWMGenIntTrigEnable().
- **ADC_TRIGGER_ALWAYS** - A trigger that is always asserted, causing the sample sequence to capture repeatedly (so long as there is not a higher priority source active).

Note that not all trigger sources are available on all Stellaris family members; consult the data sheet for the device in question to determine the availability of triggers.

The *ulPriority* parameter is a value between 0 and 3, where 0 represents the highest priority and 3 the lowest. Note that when programming the priority among a set of sample sequences, each must have unique priority; it is up to the caller to guarantee the uniqueness of the priorities.

Returns:
None.

6.2.2.10 ADCSequenceDataGet

Gets the captured data for a sample sequence.

Prototype:
```
long
ADCSequenceDataGet(unsigned long ulBase,
                   unsigned long ulSequenceNum,
                   unsigned long *pulBuffer)
```

Parameters:
ulBase is the base address of the ADC module.
ulSequenceNum is the sample sequence number.
pulBuffer is the address where the data is stored.

Description:
This function copies data from the specified sample sequence output FIFO to a memory resident buffer. The number of samples available in the hardware FIFO are copied into the buffer, which is assumed to be large enough to hold that many samples. This will only return the samples that are presently available, which may not be the entire sample sequence if it is in the process of being executed.

Returns:
Returns the number of samples copied to the buffer.

6.2.2.11 ADCSequenceDisable

Disables a sample sequence.

Prototype:
```
void
ADCSequenceDisable(unsigned long ulBase,
                   unsigned long ulSequenceNum)
```

Parameters:
ulBase is the base address of the ADC module.
ulSequenceNum is the sample sequence number.

Description:
Prevents the specified sample sequence from being captured when its trigger is detected. A sample sequence should be disabled before it is configured.

Returns:
None.

6.2.2.12 ADCSequenceEnable

Enables a sample sequence.

Prototype:

```
void
ADCSequenceEnable(unsigned long ulBase,
                  unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

Description:

Allows the specified sample sequence to be captured when its trigger is detected. A sample sequence must be configured before it is enabled.

Returns:

None.

6.2.2.13 ADCSequenceOverflow

Determines if a sample sequence overflow occurred.

Prototype:

```
long
ADCSequenceOverflow(unsigned long ulBase,
                    unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

Description:

This determines if a sample sequence overflow has occurred. This will happen if the captured samples are not read from the FIFO before the next trigger occurs.

Returns:

Returns zero if there was not an overflow, and non-zero if there was.

6.2.2.14 ADCSequenceOverflowClear

Clears the overflow condition on a sample sequence.

Prototype:

```
void
ADCSequenceOverflowClear(unsigned long ulBase,
                         unsigned long ulSequenceNum)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

Description:

This will clear an overflow condition on one of the sample sequences. The overflow condition must be cleared in order to detect a subsequent overflow condition (it otherwise causes no harm).

Returns:

None.

6.2.2.15 ADCSequenceStepConfigure

Configure a step of the sample sequencer.

Prototype:

```
void
ADCSequenceStepConfigure(unsigned long ulBase,
                         unsigned long ulSequenceNum,
                         unsigned long ulStep,
                         unsigned long ulConfig)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

ulStep is the step to be configured.

ulConfig is the configuration of this step; must be a logical OR of **ADC_CTL_TS**, **ADC_CTL_IE**, **ADC_CTL_END**, **ADC_CTL_D**, and one of the input channel selects (**ADC_CTL_CH0** through **ADC_CTL_CH7**).

Description:

This function will set the configuration of the ADC for one step of a sample sequence. The ADC can be configured for single-ended or differential operation (the **ADC_CTL_D** bit selects differential operation when set), the channel to be sampled can be chosen (the **ADC_CTL_CH0** through **ADC_CTL_CH7** values), and the internal temperature sensor can be selected (the **ADC_CTL_TS** bit). Additionally, this step can be defined as the last in the sequence (the **ADC_CTL_END** bit) and it can be configured to cause an interrupt when the step is complete (the **ADC_CTL_IE** bit). The configuration is used by the ADC at the appropriate time when the trigger for this sequence occurs.

The *ulStep* parameter determines the order in which the samples are captured by the ADC when the trigger occurs. It can range from zero to seven for the first sample sequence, from zero to three for the second and third sample sequence, and can only be zero for the fourth sample sequence.

Differential mode only works with adjacent channel pairs (for example, 0 and 1). The channel select must be the number of the channel pair to sample (for example, **ADC_CTL_CH0** for 0 and 1, or **ADC_CTL_CH1** for 2 and 3) or undefined results will be returned by the ADC. Additionally, if differential mode is selected when the temperature sensor is being sampled, undefined results will be returned by the ADC.

It is the responsibility of the caller to ensure that a valid configuration is specified; this function does not check the validity of the specified configuration.

Returns:
None.

6.2.2.16 ADCSequenceUnderflow

Determines if a sample sequence underflow occurred.

Prototype:
```
long
ADCSequenceUnderflow(unsigned long ulBase,
                     unsigned long ulSequenceNum)
```

Parameters:
ulBase is the base address of the ADC module.
ulSequenceNum is the sample sequence number.

Description:
This determines if a sample sequence underflow has occurred. This will happen if too many samples are read from the FIFO.

Returns:
Returns zero if there was not an underflow, and non-zero if there was.

6.2.2.17 ADCSequenceUnderflowClear

Clears the underflow condition on a sample sequence.

Prototype:
```
void
ADCSequenceUnderflowClear(unsigned long ulBase,
                          unsigned long ulSequenceNum)
```

Parameters:
ulBase is the base address of the ADC module.
ulSequenceNum is the sample sequence number.

Description:
This will clear an underflow condition on one of the sample sequences. The underflow condition must be cleared in order to detect a subsequent underflow condition (it otherwise causes no harm).

Returns:
None.

6.2.2.18 ADCSoftwareOversampleConfigure

Configures the software oversampling factor of the ADC.

Prototype:

```
void
ADCSoftwareOversampleConfigure(unsigned long ulBase,
                               unsigned long ulSequenceNum,
                               unsigned long ulFactor)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

ulFactor is the number of samples to be averaged.

Description:

This function configures the software oversampling for the ADC, which can be used to provide better resolution on the sampled data. Oversampling is accomplished by averaging multiple samples from the same analog input. Three different oversampling rates are supported; 2x, 4x, and 8x.

Oversampling is only supported on the sample sequencers that are more than one sample in depth (that is, the fourth sample sequencer is not supported). Oversampling by 2x (for example) divides the depth of the sample sequencer by two; so 2x oversampling on the first sample sequencer can only provide four samples per trigger. This also means that 8x oversampling is only available on the first sample sequencer.

Returns:

None.

6.2.2.19 ADCSoftwareOversampleDataGet

Gets the captured data for a sample sequence using software oversampling.

Prototype:

```
void
ADCSoftwareOversampleDataGet(unsigned long ulBase,
                             unsigned long ulSequenceNum,
                             unsigned long *pulBuffer,
                             unsigned long ulCount)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

pulBuffer is the address where the data is stored.

ulCount is the number of samples to be read.

Description:

This function copies data from the specified sample sequence output FIFO to a memory resident buffer with software oversampling applied. The requested number of samples are copied into the data buffer; if there are not enough samples in the hardware FIFO to satisfy this many oversampled data items then incorrect results will be returned. It is the caller's responsibility to read only the samples that are available and wait until enough data is available, for example as a result of receiving an interrupt.

Returns:

None.

6.2.2.20 ADCSoftwareOversampleStepConfigure

Configures a step of the software oversampled sequencer.

Prototype:

```
void
ADCSoftwareOversampleStepConfigure(unsigned long ulBase,
                                   unsigned long ulSequenceNum,
                                   unsigned long ulStep,
                                   unsigned long ulConfig)
```

Parameters:

ulBase is the base address of the ADC module.

ulSequenceNum is the sample sequence number.

ulStep is the step to be configured.

ulConfig is the configuration of this step.

Description:

This function configures a step of the sample sequencer when using the software oversampling feature. The number of steps available depends on the oversampling factor set by ADCSoftwareOversampleConfigure(). The value of *ulConfig* is the same as defined for ADCSequenceStepConfigure().

Returns:

None.

6.3 Programming Example

The following example shows how to use the ADC API to initialize a sample sequence for processor triggering, trigger the sample sequence, and then read back the data when it is ready.

```
unsigned long ulValue;

//
// Enable the first sample sequence to capture the value of channel 0 when
// the processor trigger occurs.
//
ADCSequenceConfigure(ADC_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
ADCSequenceStepConfigure(ADC_BASE, 0, 0,
                         ADC_CTL_IE | ADC_CTL_END | ADC_CTL_CH0);
ADCSequenceEnable(ADC_BASE, 0);

//
// Trigger the sample sequence.
//
ADCProcessorTrigger(ADC_BASE, 0);

//
// Wait until the sample sequence has completed.
//
while(!ADCIntStatus(ADC_BASE, 0, false))
{
}

//
```

```
// Read the value from the ADC.
//
ADCSequenceDataGet(ADC_BASE, 0, &ulValue);
```

7 Controller Area Network (CAN)

7.1 Introduction

The Controller Area Network (CAN) APIs provide a set of functions for accessing the Stellaris CAN modules. Functions are provided to configure the CAN controllers, configure message objects, and manage CAN interrupts.

The Stellaris CAN module provides hardware processing of the CAN data link layer. It can be configured with message filters and preloaded message data so that it can autonomously send and receive messages on the bus, and notify the application accordingly. It automatically handles generation and checking of CRCs, error processing, and retransmission of CAN messages.

The message objects are stored in the CAN controller and provide the main interface for the CAN module on the CAN bus. There are 32 message objects that can each be programmed to handle a separate message ID, or can be chained together for a sequence of frames with the same ID. The message identifier filters provide masking that can be programmed to match any or all of the message ID bits, and frame types.

This driver is contained in `src/can.c`, with `src/can.h` containing the API definitions for use by applications.

7.2 API Functions

Data Structures

- tCANBitClkParms
- tCANMsgObject

Defines

- MSG_OBJ_STATUS_MASK

Enumerations

- tCANIntFlags
- tCANIntStsReg
- tCANObjFlags
- tCANStatusCtrl
- tCANStsReg
- tMsgObjType

Functions

- void CANBitTimingGet (unsigned long ulBase, tCANBitClkParms *pClkParms)
- void CANBitTimingSet (unsigned long ulBase, tCANBitClkParms *pClkParms)
- void CANDisable (unsigned long ulBase)
- void CANEnable (unsigned long ulBase)
- tBoolean CANErrCntrGet (unsigned long ulBase, unsigned long *pulRxCount, unsigned long *pulTxCount)
- void CANInit (unsigned long ulBase)
- void CANIntClear (unsigned long ulBase, unsigned long ulIntClr)
- void CANIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void CANIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- void CANIntRegister (unsigned long ulBase, void (*pfnHandler)(void))
- unsigned long CANIntStatus (unsigned long ulBase, tCANIntStsReg eIntStsReg)
- void CANIntUnregister (unsigned long ulBase)
- void CANMessageClear (unsigned long ulBase, unsigned long ulObjID)
- void CANMessageGet (unsigned long ulBase, unsigned long ulObjID, tCANMsgObject *pMsgObject, tBoolean bClrPendingInt)
- void CANMessageSet (unsigned long ulBase, unsigned long ulObjID, tCANMsgObject *pMsgObject, tMsgObjType eMsgType)
- tBoolean CANRetryGet (unsigned long ulBase)
- void CANRetrySet (unsigned long ulBase, tBoolean bAutoRetry)
- unsigned long CANStatusGet (unsigned long ulBase, tCANStsReg eStatusReg)

7.2.1 Detailed Description

The CAN APIs provide all of the functions needed by the application to implement an interrupt-driven CAN stack. These functions may be used to control any of the available CAN ports on a Stellaris microcontroller, and can be used with one port without causing conflicts with the other port.

The CAN module is disabled by default, so the the CANInit() function must be called before any other CAN functions are called. This call initializes the message objects to a safe state prior to enabling the controller on the CAN bus. Also, the bit timing values must be programmed prior to enabling the CAN controller. The CANSetBitTiming() function should be called with the appropriate bit timing values for the CAN bus. Once these two functions have been called, a CAN controller can be enabled using the CANEnable(), and later disabled using CANDisable() if needed. Calling CANDisable() does not reinitialize a CAN controller, so it can be used to temporarily remove a CAN controller from the bus.

The CAN controller is highly configurable and contains 32 message objects that can be programmed to automatically transmit and receive CAN messages under certain conditions. Message objects allow the application to perform some actions automatically without interaction from the microcontroller. Some examples of these actions are the following:

- Send a data frame immediately
- Send a data frame when a matching remote frame is seen on the CAN bus
- Receive a specific data frame
- Receive data frames that match a certain identifier pattern

To configure message objects to perform any of these actions, the application must first set up one of the 32 message objects using CANMessageSet(). This function must be used to configure a message object to send data, or to configure a message object to receive data. Each message object can be configured to generate interrupts on transmission or reception of CAN messages.

When data is received from the CAN bus, the application can use the CANMessageGet() function to read the received message. This function can also be used to read a message object that is already configured in order to populate a message structure prior to making changes to the configuration of a message object. Reading the message object using this function will also clear any pending interrupt on the message object.

Once a message object has been configured using CANMessageSet(), it has allocated the message object and will continue to perform its programmed function unless it is released with a call to CANMessageClear(). The application is not required to clear out a message object before setting it with a new configuration, because each time CANMessageSet() is called, it will overwrite any previously programmed configuration.

The 32 message objects are identical except for priority. The lowest numbered message objects have the highest priority. Priority affects operation in two ways. First, if multiple actions are ready at the same time, the one with the highest priority message object will occur first. And second, when multiple message objects have interrupts pending, the highest priority will be presented first when reading the interrupt status. It is up to the application to manage the 32 message objects as a resource, and determine the best method for allocating and releasing them.

The CAN controller can generate interrupts on several conditions:

- When any message object transmits a message
- When any message object receives a message
- On warning conditions such as an error counter reaching a limit or occurrence of various bus errors
- On controller error conditions such as entering the bus-off state

An interrupt handler must be installed in order to process CAN interrupts. If dynamic interrupt configuration is desired, the CANIntRegister() can be used to register the interrupt handler. This will place the vector in a RAM-based vector table. However, if the application uses a pre-loaded vector table in flash, then the CAN controller handler should be entered in the appropriate slot in the vector table. In this case, CANIntRegister() is not needed, but the interrupt will need to be enabled on the host processor master interrupt controller using the IntEnable() function. The CAN module interrupts are enabled using the CANIntEnable() function. They can be disabled by using the CANIntDisable() function.

Once CAN interrupts are enabled, the handler will be invoked whenever a CAN interrupt is triggered. The handler can determine which condition caused the interrupt by using the CANIntStatus() function. Multiple conditions can be pending when an interrupt occurs, so the handler must be designed to process all pending interrupt conditions before exiting. Each interrupt condition must be cleared before exiting the handler. There are two ways to do this. The CANIntClear() function will clear a specific interrupt condition without further action required by the handler. However, the handler can also clear the condition by performing certain actions. If the interrupt is a status interrupt, the interrupt can be cleared by reading the status register with CANStatusGet(). If the interrupt is caused by one of the message objects, then it can be cleared by reading the message object using CANMessageGet().

There are several status registers that can be used to help the application manage the controller. The status registers are read using the CANStatusGet() function. There is a controller status register that provides general status information such as error or warning conditions. There are also

several status registers that provide information about all of the message objects at once using a 32-bit bit map of the status, with one bit representing each message object. These status registers can be used to determine:

- Which message objects have unprocessed received data
- Which message objects have pending transmission requests
- Which message objects are allocated for use

7.2.2 Data Structure Documentation

7.2.2.1 tCANBitClkParms

Definition:

```
typedef struct
{
    unsigned int uSyncPropPhase1Seg;
    unsigned int uPhase2Seg;
    unsigned int uSJW;
    unsigned int uQuantumPrescaler;
}
tCANBitClkParms
```

Members:

uSyncPropPhase1Seg This value holds the sum of the Synchronization, Propagation, and Phase Buffer 1 segments, measured in time quanta. The valid values for this setting range from 2 to 16.

uPhase2Seg This value holds the Phase Buffer 2 segment in time quanta. The valid values for this setting range from 1 to 8.

uSJW This value holds the Resynchronization Jump Width in time quanta. The valid values for this setting range from 1 to 4.

uQuantumPrescaler This value holds the CAN_CLK divider used to determine time quanta. The valid values for this setting range from 1 to 1023.

Description:

This structure is used for encapsulating the values associated with setting up the bit timing for a CAN controller. The structure is used when calling the CANGetBitTiming and CANSetBitTiming functions.

7.2.2.2 tCANMsgObject

Definition:

```
typedef struct
{
    unsigned long ulMsgID;
    unsigned long ulMsgIDMask;
    unsigned long ulFlags;
    unsigned long ulMsgLen;
    unsigned char *pucMsgData;
}
tCANMsgObject
```

Members:

ulMsgID The CAN message identifier used for 11 or 29 bit identifiers.

ulMsgIDMask The message identifier mask used when identifier filtering is enabled.

ulFlags This value holds various status flags and settings specified by tCANObjFlags.

ulMsgLen This value is the number of bytes of data in the message object.

pucMsgData This is a pointer to the message object's data.

Description:

The structure used for encapsulating all the items associated with a CAN message object in the CAN controller.

7.2.3 Define Documentation

7.2.3.1 MSG_OBJ_STATUS_MASK

Definition:

```
#define MSG_OBJ_STATUS_MASK
```

Description:

This define is used with the tCANObjFlags enumerated values to allow checking only status flags and not configuration flags.

7.2.4 Enumeration Documentation

7.2.4.1 tCANIntFlags

Description:

These definitions are used to specify interrupt sources to CANIntEnable() and CANIntDisable().

Enumerators:

CAN_INT_ERROR This flag is used to allow a CAN controller to generate error interrupts.

CAN_INT_STATUS This flag is used to allow a CAN controller to generate status interrupts.

CAN_INT_MASTER This flag is used to allow a CAN controller to generate any CAN interrupts. If this is not set, then no interrupts will be generated by the CAN controller.

7.2.4.2 tCANIntStsReg

Description:

This data type is used to identify the interrupt status register. This is used when calling the CANIntStatus() function.

Enumerators:

CAN_INT_STS_CAUSE Read the CAN interrupt status information.

CAN_INT_STS_OBJECT Read a message object's interrupt status.

7.2.4.3 tCANObjFlags

Description:

These are the flags used by the tCANMsgObject variable when calling the CANMessageSet() and CANMessageGet() functions.

Enumerators:

MSG_OBJ_TX_INT_ENABLE This indicates that transmit interrupts should be enabled, or are enabled.

MSG_OBJ_RX_INT_ENABLE This indicates that receive interrupts should be enabled, or are enabled.

MSG_OBJ_EXTENDED_ID This indicates that a message object will use or is using an extended identifier.

MSG_OBJ_USE_ID_FILTER This indicates that a message object will use or is using filtering based on the object's message identifier.

MSG_OBJ_NEW_DATA This indicates that new data was available in the message object.

MSG_OBJ_DATA_LOST This indicates that data was lost since this message object was last read.

MSG_OBJ_USE_DIR_FILTER This indicates that a message object will use or is using filtering based on the direction of the transfer. If the direction filtering is used, then ID filtering must also be enabled.

MSG_OBJ_USE_EXT_FILTER This indicates that a message object will use or is using message identifier filtering based on the extended identifier. If the extended identifier filtering is used, then ID filtering must also be enabled.

MSG_OBJ_REMOTE_FRAME This indicates that a message object is a remote frame.

MSG_OBJ_NO_FLAGS This indicates that a message object has no flags set.

7.2.4.4 tCANStatusCtrl

Description:

The following enumeration contains all error or status indicators that can be returned when calling the CANStatusGet() function.

Enumerators:

CAN_STATUS_BUS_OFF CAN controller has entered a Bus Off state.

CAN_STATUS_EWARN CAN controller error level has reached warning level.

CAN_STATUS_EPASS CAN controller error level has reached error passive level.

CAN_STATUS_RXOK A message was received successfully since the last read of this status.

CAN_STATUS_TXOK A message was transmitted successfully since the last read of this status.

CAN_STATUS_LEC_MSK This is the mask for the last error code field.

CAN_STATUS_LEC_NONE There was no error.

CAN_STATUS_LEC_STUFF A bit stuffing error has occurred.

CAN_STATUS_LEC_FORM A formatting error has occurred.

CAN_STATUS_LEC_ACK An acknowledge error has occurred.

CAN_STATUS_LEC_BIT1 The bus remained a bit level of 1 for longer than is allowed.

CAN_STATUS_LEC_BIT0 The bus remained a bit level of 0 for longer than is allowed.

CAN_STATUS_LEC_CRC A CRC error has occurred.
CAN_STATUS_LEC_MASK This is the mask for the CAN Last Error Code (LEC).

7.2.4.5 tCANStsReg

Description:
This data type is used to identify which of several status registers to read when calling the CANStatusGet() function.

Enumerators:
CAN_STS_CONTROL Read the full CAN controller status.
CAN_STS_TXREQUEST Read the full 32-bit mask of message objects with a transmit request set.
CAN_STS_NEWDAT Read the full 32-bit mask of message objects with new data available.
CAN_STS_MSGVAL Read the full 32-bit mask of message objects that are enabled.

7.2.4.6 tMsgObjType

Description:
This definition is used to determine the type of message object that will be set up via a call to the CANMessageSet() API.

Enumerators:
MSG_OBJ_TYPE_TX Transmit message object.
MSG_OBJ_TYPE_TX_REMOTE Transmit remote request message object.
MSG_OBJ_TYPE_RX Receive message object.
MSG_OBJ_TYPE_RX_REMOTE Receive remote request message object.
MSG_OBJ_TYPE_RXTX_REMOTE Remote frame receive remote, with auto-transmit message object.

7.2.5 Function Documentation

7.2.5.1 CANBitTimingGet

Reads the current settings for the CAN controller bit timing.

Prototype:
```
void
CANBitTimingGet(unsigned long ulBase,
                tCANBitClkParms *pClkParms)
```

Parameters:
ulBase is the base address of the CAN controller.
pClkParms is a pointer to a structure to hold the timing parameters.

Description:

This function reads the current configuration of the CAN controller bit clock timing, and stores the resulting information in the structure supplied by the caller. Refer to CANBitTimingSet() for the meaning of the values that are returned in the structure pointed to by *pClkParms*.

This function replaces the original CANGetBitTiming() API and performs the same actions. A macro is provided in `can.h` to map the original API to this API.

Returns:

None.

7.2.5.2 CANBitTimingSet

Configures the CAN controller bit timing.

Prototype:

```
void
CANBitTimingSet(unsigned long ulBase,
                tCANBitClkParms *pClkParms)
```

Parameters:

ulBase is the base address of the CAN controller.

pClkParms points to the structure with the clock parameters.

Description:

Configures the various timing parameters for the CAN bus bit timing: Propagation segment, Phase Buffer 1 segment, Phase Buffer 2 segment, and the Synchronization Jump Width. The values for Propagation and Phase Buffer 1 segments are derived from the combination *pClkParms->uSyncPropPhase1Seg* parameter. Phase Buffer 2 is determined from the *pClkParms->uPhase2Seg* parameter. These two parameters, along with *pClkParms->uSJW* are based in units of bit time quanta. The actual quantum time is determined by the *pClkParms->uQuantumPrescaler* value, which specifies the divisor for the CAN module clock.

The total bit time, in quanta, will be the sum of the two Seg parameters, as follows:

bit_time_q = uSyncPropPhase1Seg + uPhase2Seg + 1

Note that the Sync_Seg is always one quantum in duration, and will be added to derive the correct duration of Prop_Seg and Phase1_Seg.

The equation to determine the actual bit rate is as follows:

CAN Clock / ((*uSyncPropPhase1Seg* + *uPhase2Seg* + 1) $*$ (*uQuantumPrescaler*))

This means that with *uSyncPropPhase1Seg* = 4, *uPhase2Seg* = 1, *uQuantumPrescaler* = 2 and an 8 MHz CAN clock, that the bit rate will be (8 MHz) / ((5 + 2 + 1) $*$ 2) or 500 Kbit/sec.

This function replaces the original CANSetBitTiming() API and performs the same actions. A macro is provided in `can.h` to map the original API to this API.

Returns:

None.

7.2.5.3 CANDisable

Disables the CAN controller.

Prototype:

```
void
CANDisable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the CAN controller to disable.

Description:

Disables the CAN controller for message processing. When disabled, the controller will no longer automatically process data on the CAN bus. The controller can be restarted by calling CANEnable(). The state of the CAN controller and the message objects in the controller are left as they were before this call was made.

Returns:

None.

7.2.5.4 CANEnable

Enables the CAN controller.

Prototype:

```
void
CANEnable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the CAN controller to enable.

Description:

Enables the CAN controller for message processing. Once enabled, the controller will automatically transmit any pending frames, and process any received frames. The controller can be stopped by calling CANDisable(). Prior to calling CANEnable(), CANInit() should have been called to initialize the controller and the CAN bus clock should be configured by calling CANBitTimingSet().

Returns:

None.

7.2.5.5 CANErrCntrGet

Reads the CAN controller error counter register.

Prototype:

```
tBoolean
CANErrCntrGet(unsigned long ulBase,
              unsigned long *pulRxCount,
              unsigned long *pulTxCount)
```

Parameters:

ulBase is the base address of the CAN controller.

pulRxCount is a pointer to storage for the receive error counter.

pulTxCount is a pointer to storage for the transmit error counter.

Description:

Reads the error counter register and returns the transmit and receive error counts to the caller along with a flag indicating if the controller receive counter has reached the error passive limit. The values of the receive and transmit error counters are returned through the pointers provided as parameters.

After this call, $*$*pulRxCount* will hold the current receive error count and $*$*pulTxCount* will hold the current transmit error count.

Returns:

Returns **true** if the receive error count has reached the error passive limit, and **false** if the error count is below the error passive limit.

7.2.5.6 CANInit

Initializes the CAN controller after reset.

Prototype:

```
void
CANInit(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the CAN controller.

Description:

After reset, the CAN controller is left in the disabled state. However, the memory used for message objects contains undefined values and must be cleared prior to enabling the CAN controller the first time. This prevents unwanted transmission or reception of data before the message objects are configured. This function must be called before enabling the controller the first time.

Returns:

None.

7.2.5.7 CANIntClear

Clears a CAN interrupt source.

Prototype:

```
void
CANIntClear(unsigned long ulBase,
            unsigned long ulIntClr)
```

Parameters:

ulBase is the base address of the CAN controller.

ulIntClr is a value indicating which interrupt source to clear.

Description:

This function can be used to clear a specific interrupt source. The *ulIntClr* parameter should be one of the following values:

- **CAN_INT_INTID_STATUS** - Clears a status interrupt.
- 1-32 - Clears the specified message object interrupt

It is not necessary to use this function to clear an interrupt. This should only be used if the application wants to clear an interrupt source without taking the normal interrupt action.

Normally, the status interrupt is cleared by reading the controller status using CANStatusGet(). A specific message object interrupt is normally cleared by reading the message object using CANMessageGet().

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

7.2.5.8 CANIntDisable

Disables individual CAN controller interrupt sources.

Prototype:

```
void
CANIntDisable(unsigned long ulBase,
              unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the CAN controller.

ulIntFlags is the bit mask of the interrupt sources to be disabled.

Description:

Disables the specified CAN controller interrupt sources. Only enabled interrupt sources can cause a processor interrupt.

The *ulIntFlags* parameter has the same definition as in the CANIntEnable() function.

Returns:

None.

7.2.5.9 CANIntEnable

Enables individual CAN controller interrupt sources.

Prototype:

```
void
CANIntEnable(unsigned long ulBase,
             unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the CAN controller.

ulIntFlags is the bit mask of the interrupt sources to be enabled.

Description:

Enables specific interrupt sources of the CAN controller. Only enabled sources will cause a processor interrupt.

The *ulIntFlags* parameter is the logical OR of any of the following:

- **CAN_INT_ERROR** - a controller error condition has occurred
- **CAN_INT_STATUS** - a message transfer has completed, or a bus error has been detected
- **CAN_INT_MASTER** - allow CAN controller to generate interrupts

In order to generate any interrupts, **CAN_INT_MASTER** must be enabled. Further, for any particular transaction from a message object to generate an interrupt, that message object must have interrupts enabled (see CANMessageSet()). **CAN_INT_ERROR** will generate an interrupt if the controller enters the “bus off” condition, or if the error counters reach a limit. **CAN_INT_STATUS** will generate an interrupt under quite a few status conditions and may provide more interrupts than the application needs to handle. When an interrupt occurs, use CANIntStatus() to determine the cause.

Returns:

None.

7.2.5.10 CANIntRegister

Registers an interrupt handler for the CAN controller.

Prototype:

```
void
CANIntRegister(unsigned long ulBase,
               void (*pfnHandler)(void))
```

Parameters:

ulBase is the base address of the CAN controller.

pfnHandler is a pointer to the function to be called when the enabled CAN interrupts occur.

Description:

This function registers the interrupt handler in the interrupt vector table, and enables CAN interrupts on the interrupt controller; specific CAN interrupt sources must be enabled using CANIntEnable(). The interrupt handler being registered must clear the source of the interrupt using CANIntClear().

If the application is using a static interrupt vector table stored in flash, then it is not necessary to register the interrupt handler this way. Instead, IntEnable() should be used to enable CAN interrupts on the interrupt controller.

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

7.2.5.11 CANIntStatus

Returns the current CAN controller interrupt status.

Prototype:
```
unsigned long
CANIntStatus(unsigned long ulBase,
             tCANIntStsReg eIntStsReg)
```

Parameters:
ulBase is the base address of the CAN controller.
eIntStsReg indicates which interrupt status register to read

Description:
Returns the value of one of two interrupt status registers. The interrupt status register read is determined by the *eIntStsReg* parameter, which can have one of the following values:

- **CAN_INT_STS_CAUSE** - indicates the cause of the interrupt
- **CAN_INT_STS_OBJECT** - indicates pending interrupts of all message objects

CAN_INT_STS_CAUSE returns the value of the controller interrupt register and indicates the cause of the interrupt. It will be a value of **CAN_INT_INTID_STATUS** if the cause is a status interrupt. In this case, the status register should be read with the CANStatusGet() function. Calling this function to read the status will also clear the status interrupt. If the value of the interrupt register is in the range 1-32, then this indicates the number of the highest priority message object that has an interrupt pending. The message object interrupt can be cleared by using the CANIntClear() function, or by reading the message using CANMessageGet() in the case of a received message. The interrupt handler can read the interrupt status again to make sure all pending interrupts are cleared before returning from the interrupt.

CAN_INT_STS_OBJECT returns a bit mask indicating which message objects have pending interrupts. This can be used to discover all of the pending interrupts at once, as opposed to repeatedly reading the interrupt register by using **CAN_INT_STS_CAUSE**.

Returns:
Returns the value of one of the interrupt status registers.

7.2.5.12 CANIntUnregister

Unregisters an interrupt handler for the CAN controller.

Prototype:
```
void
CANIntUnregister(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the controller.

Description:

This function unregisters the previously registered interrupt handler and disables the interrupt on the interrupt controller.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

7.2.5.13 CANMessageClear

Clears a message object so that it is no longer used.

Prototype:

```
void
CANMessageClear(unsigned long ulBase,
                unsigned long ulObjID)
```

Parameters:

ulBase is the base address of the CAN controller.

ulObjID is the message object number to disable (1-32).

Description:

This function frees the specified message object from use. Once a message object has been "cleared," it will no longer automatically send or receive messages, or generate interrupts.

Returns:

None.

7.2.5.14 CANMessageGet

Reads a CAN message from one of the message object buffers.

Prototype:

```
void
CANMessageGet(unsigned long ulBase,
              unsigned long ulObjID,
              tCANMsgObject *pMsgObject,
              tBoolean bClrPendingInt)
```

Parameters:

ulBase is the base address of the CAN controller.

ulObjID is the object number to read (1-32).

pMsgObject points to a structure containing message object fields.

bClrPendingInt indicates whether an associated interrupt should be cleared.

Description:

This function is used to read the contents of one of the 32 message objects in the CAN controller, and return it to the caller. The data returned is stored in the fields of the caller-supplied structure pointed to by *pMsgObject*. The data consists of all of the parts of a CAN message, plus some control and status information.

Normally this is used to read a message object that has received and stored a CAN message with a certain identifier. However, this could also be used to read the contents of a message object in order to load the fields of the structure in case only part of the structure needs to be changed from a previous setting.

When using CANMessageGet, all of the same fields of the structure are populated in the same way as when the CANMessageSet() function is used, with the following exceptions:

pMsgObject->ulFlags:

- **MSG_OBJ_NEW_DATA** indicates if this is new data since the last time it was read
- **MSG_OBJ_DATA_LOST** indicates that at least one message was received on this message object, and not read by the host before being overwritten.

Returns:

None.

7.2.5.15 CANMessageSet

Configures a message object in the CAN controller.

Prototype:

```
void
CANMessageSet(unsigned long ulBase,
              unsigned long ulObjID,
              tCANMsgObject *pMsgObject,
              tMsgObjType eMsgType)
```

Parameters:

ulBase is the base address of the CAN controller.

ulObjID is the object number to configure (1-32).

pMsgObject is a pointer to a structure containing message object settings.

eMsgType indicates the type of message for this object.

Description:

This function is used to configure any one of the 32 message objects in the CAN controller. A message object can be configured as any type of CAN message object as well as several options for automatic transmission and reception. This call also allows the message object to be configured to generate interrupts on completion of message receipt or transmission. The message object can also be configured with a filter/mask so that actions are only taken when a message that meets certain parameters is seen on the CAN bus.

The *eMsgType* parameter must be one of the following values:

- **MSG_OBJ_TYPE_TX** - CAN transmit message object.
- **MSG_OBJ_TYPE_TX_REMOTE** - CAN transmit remote request message object.
- **MSG_OBJ_TYPE_RX** - CAN receive message object.

- **MSG_OBJ_TYPE_RX_REMOTE** - CAN receive remote request message object.
- **MSG_OBJ_TYPE_RXTX_REMOTE** - CAN remote frame receive remote, then transmit message object.

The message object pointed to by *pMsgObject* must be populated by the caller, as follows:

- *ulMsgID* - contains the message ID, either 11 or 29 bits.
- *ulMsgIDMask* - mask of bits from *ulMsgID* that must match if identifier filtering is enabled.
- *ulFlags*
  - Set **MSG_OBJ_TX_INT_ENABLE** flag to enable interrupt on transmission.
  - Set **MSG_OBJ_RX_INT_ENABLE** flag to enable interrupt on receipt.
  - Set **MSG_OBJ_USE_ID_FILTER** flag to enable filtering based on the identifier mask specified by *ulMsgIDMask*.
- *ulMsgLen* - the number of bytes in the message data. This should be non-zero even for a remote frame; it should match the expected bytes of the data responding data frame.
- *pucMsgData* - points to a buffer containing up to 8 bytes of data for a data frame.

Example: To send a data frame or remote frame(in response to a remote request), take the following steps:

1. Set *eMsgType* to **MSG_OBJ_TYPE_TX**.
2. Set *pMsgObject->ulMsgID* to the message ID.
3. Set *pMsgObject->ulFlags*. Make sure to set **MSG_OBJ_TX_INT_ENABLE** to allow an interrupt to be generated when the message is sent.
4. Set *pMsgObject->ulMsgLen* to the number of bytes in the data frame.
5. Set *pMsgObject->pucMsgData* to point to an array containing the bytes to send in the message.
6. Call this function with *ulObjID* set to one of the 32 object buffers.

Example: To receive a specific data frame, take the following steps:

1. Set *eMsgObjType* to **MSG_OBJ_TYPE_RX**.
2. Set *pMsgObject->ulMsgID* to the full message ID, or a partial mask to use partial ID matching.
3. Set *pMsgObject->ulMsgIDMask* bits that should be used for masking during comparison.
4. Set *pMsgObject->ulFlags* as follows:
   - Set **MSG_OBJ_TX_INT_ENABLE** flag to be interrupted when the data frame is received.
   - Set **MSG_OBJ_USE_ID_FILTER** flag to enable identifier based filtering.
5. Set *pMsgObject->ulMsgLen* to the number of bytes in the expected data frame.
6. The buffer pointed to by *pMsgObject->pucMsgData* and *pMsgObject->ulMsgLen* are not used by this call as no data is present at the time of the call.
7. Call this function with *ulObjID* set to one of the 32 object buffers.

If you specify a message object buffer that already contains a message definition, it will be overwritten.

Returns:

None.

7.2.5.16 CANRetryGet

Returns the current setting for automatic retransmission.

Prototype:

```
tBoolean
CANRetryGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the CAN controller.

Description:

Reads the current setting for the automatic retransmission in the CAN controller and returns it to the caller.

Returns:

Returns **true** if automatic retransmission is enabled, **false** otherwise.

7.2.5.17 CANRetrySet

Sets the CAN controller automatic retransmission behavior.

Prototype:

```
void
CANRetrySet(unsigned long ulBase,
            tBoolean bAutoRetry)
```

Parameters:

ulBase is the base address of the CAN controller.

bAutoRetry enables automatic retransmission.

Description:

Enables or disables automatic retransmission of messages with detected errors. If *bAutoRetry* is **true**, then automatic retransmission is enabled, otherwise it is disabled.

Returns:

None.

7.2.5.18 CANStatusGet

Reads one of the controller status registers.

Prototype:

```
unsigned long
CANStatusGet(unsigned long ulBase,
             tCANStsReg eStatusReg)
```

Parameters:

ulBase is the base address of the CAN controller.

eStatusReg is the status register to read.

Description:

Reads a status register of the CAN controller and returns it to the caller. The different status registers are:

- **CAN_STS_CONTROL** - the main controller status
- **CAN_STS_TXREQUEST** - bit mask of objects pending transmission
- **CAN_STS_NEWDAT** - bit mask of objects with new data
- **CAN_STS_MSGVAL** - bit mask of objects with valid configuration

When reading the main controller status register, a pending status interrupt will be cleared. This should be used in the interrupt handler for the CAN controller if the cause is a status interrupt. The controller status register fields are as follows:

- **CAN_STATUS_BUS_OFF** - controller is in bus-off condition
- **CAN_STATUS_EWARN** - an error counter has reached a limit of at least 96
- **CAN_STATUS_EPASS** - CAN controller is in the error passive state
- **CAN_STATUS_RXOK** - a message was received successfully (independent of any message filtering).
- **CAN_STATUS_TXOK** - a message was successfully transmitted
- **CAN_STATUS_LEC_MSK** - mask of last error code bits (3 bits)
- **CAN_STATUS_LEC_NONE** - no error
- **CAN_STATUS_LEC_STUFF** - stuffing error detected
- **CAN_STATUS_LEC_FORM** - a format error occurred in the fixed format part of a message
- **CAN_STATUS_LEC_ACK** - a transmitted message was not acknowledged
- **CAN_STATUS_LEC_BIT1** - dominant level detected when trying to send in recessive mode
- **CAN_STATUS_LEC_BIT0** - recessive level detected when trying to send in dominant mode
- **CAN_STATUS_LEC_CRC** - CRC error in received message

The remaining status registers are 32-bit bit maps to the message objects. They can be used to quickly obtain information about the status of all the message objects without needing to query each one. They contain the following information:

- **CAN_STS_TXREQUEST** - if a message object's TxRequest bit is set, that means that a transmission is pending on that object. The application can use this to determine which objects are still waiting to send a message.
- **CAN_STS_NEWDAT** - if a message object's NewDat bit is set, that means that a new message has been received in that object, and has not yet been picked up by the host application
- **CAN_STS_MSGVAL** - if a message object's MsgVal bit is set, that means it has a valid configuration programmed. The host application can use this to determine which message objects are empty/unused.

Returns:

Returns the value of the status register.

7.3 Programming Example

This example code will send out data from CAN controller 0 to be received by CAN controller 1. In order to actually receive the data, an external cable must be connected between the two ports. In

this example, both controllers are configured for 1 Mbit operation.

```
tCANBitClkParms CANBitClk;
tCANMsgObject sMsgObjectRx;
tCANMsgObject sMsgObjectTx;
unsigned char ucBufferIn[8];
unsigned char ucBufferOut[8];

//
// Reset the state of all the message objects and the state of the CAN
// module to a known state.
//
CANInit(CAN0_BASE);
CANInit(CAN1_BASE);

//
// Configure the controller for 1 Mbit operation.
//
CANBitClk.uSyncPropPhase1Seg = 5;
CANBitClk.uPhase2Seg = 2;
CANBitClk.uQuantumPrescaler = 1;
CANBitClk.uSJW = 2;
CANSetBitTiming(CAN0_BASE, &CANBitClk);
CANSetBitTiming(CAN1_BASE, &CANBitClk);

//
// Take the CAN0 device out of INIT state.
//
CANEnable(CAN0_BASE);
CANEnable(CAN1_BASE);

//
// Configure a receive object.
//
sMsgObjectRx.ulMsgID = (0x400);
sMsgObjectRx.ulMsgIDMask = 0x7f8;
sMsgObjectRx.ulFlags = MSG_OBJ_USE_ID_FILTER;
sMsgObjectRx.ulMsgLen = 8;
sMsgObjectRx.pucMsgData = ucBufferIn;
CANMessageSet(CAN1_BASE, 1, &sMsgObjectRx, MSG_OBJ_TYPE_RX);

//
// Configure and start transmit of message object.
//
sMsgObjectTx.ulMsgID = 0x400;
sMsgObjectTx.ulFlags = 0;
sMsgObjectTx.ulMsgLen = 8;
sMsgObjectTx.pucMsgData = ucBufferOut;
CANMessageSet(CAN0_BASE, 2, &sMsgObjectTx, MSG_OBJ_TYPE_TX);

//
// Wait for new data to become available.
//
while((CANStatusGet(CAN1_BASE, CAN_STS_NEWDAT) & 1) == 0)
{
    //
    // Read the message out of the message object.
    //
    CANMessageGet(CAN1_BASE, 1, &sMsgObjectRx, true);
}

//
// Process new data in sMsgObjectRx.pucMsgData.
//
...
```

8 Ethernet Controller

8.1 Introduction

The Stellaris Ethernet controller consists of a fully integrated media access controller (MAC) and a network physical (PHY) interface device. The Ethernet controller conforms to IEEE 802.3 specifications and fully supports 10BASE-T and 100BASE-TX standards.

The Ethernet API provides the set of functions required to implement an interrupt-driven Ethernet driver for this Ethernet controller. Functions are provided to configure and control the MAC, to access the register set on the PHY, to transmit and receive Ethernet packets, and to configure and control the interrupts that are available.

This driver is contained in `src/ethernet.c`, with `src/ethernet.h` containing the API definitions for use by applications.

8.2 API Functions

Functions

- unsigned long EthernetConfigGet (unsigned long ulBase)
- void EthernetConfigSet (unsigned long ulBase, unsigned long ulConfig)
- void EthernetDisable (unsigned long ulBase)
- void EthernetEnable (unsigned long ulBase)
- void EthernetInitExpClk (unsigned long ulBase, unsigned long ulEthClk)
- void EthernetIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void EthernetIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void EthernetIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- void EthernetIntRegister (unsigned long ulBase, void (*pfnHandler)(void))
- unsigned long EthernetIntStatus (unsigned long ulBase, tBoolean bMasked)
- void EthernetIntUnregister (unsigned long ulBase)
- void EthernetMACAddrGet (unsigned long ulBase, unsigned char *pucMACAddr)
- void EthernetMACAddrSet (unsigned long ulBase, unsigned char *pucMACAddr)
- tBoolean EthernetPacketAvail (unsigned long ulBase)
- long EthernetPacketGet (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)
- long EthernetPacketGetNonBlocking (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)
- long EthernetPacketPut (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)
- long EthernetPacketPutNonBlocking (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)
- unsigned long EthernetPHYRead (unsigned long ulBase, unsigned char ucRegAddr)

- void EthernetPHYWrite (unsigned long ulBase, unsigned char ucRegAddr, unsigned long ulData)
- tBoolean EthernetSpaceAvail (unsigned long ulBase)

8.2.1 Detailed Description

For any application, the EthernetInitExpClk() function must be called first to prepare the Ethernet controller for operation. This function will configure the Ethernet controller options that are based on system parameters, such as the system clock speed.

Once initialized, access to the PHY is available via the EthernetPHYRead() and EthernetPHYWrite() functions. By default, the PHY will auto-negotiate the line speed and duplex modes. For most applications, this will be sufficient. If a special configuration is required, the PHY read and write functions can be used to reconfigure the PHY to the desired mode of operation.

The MAC must also be configured using the EthernetConfigSet() function. The parameters for this function will allow the configuration of options such as Promiscuous Mode, Multicast Reception, Transmit Data Length Padding, and so on. The EthernetConfigGet() function can be used to query the current configuration of the Ethernet MAC.

The MAC address, used for incoming packet filtering, must also be programmed using the EthernetMACAddrSet() function. The current value can be queried using the EthernetMACAddrGet() function.

When configuration has been completed, the Ethernet controller can be enabled using the EthernetEnable() function. When getting ready to terminate operations on the Ethernet controller, the EthernetDisable() function may be called.

After the Ethernet controller has been enabled, Ethernet frames can be transmitted and received using the EthernetPacketPut() and EthernetPacketGet() functions. Care must be taken when using these functions, as they are blocking functions, and will not return until data is available (for RX) or buffer space is available (for TX). The EthernetSpaceAvail() and EthernetPacketAvail() functions can be called to determine if there is room for a TX packet or if there is an RX packet available prior to calling these blocking functions. Alternatively, the EthernetPacketGetNonBlocking() and EthernetPacketPutNonBlocking() functions will return immediately if a packet cannot be processed. Otherwise, the packet will be processed normally.

When developing a mapping layer for a TCP/IP stack, you may wish to use the interrupt capability of the Ethernet controller. The EthernetIntRegister() and EthernetIntUnregister() functions are used to register an ISR with the system and to enable or disable the Ethernet controller's interrupt signal. The EthernetIntEnable() and EthernetIntDisable() functions are used to manipulate the individual interrupt sources available in the Ethernet controller (for example, RX Error, TX Complete). The EthernetIntStatus() and EthernetIntClear() functions would be used to query the active interrupts to determine which process to service, and to clear the indicated interrupts prior to returning from the registered ISR.

The EthernetInit(), EthernetPacketNonBlockingGet(), and EthernetPacketNonBlockingPut() APIs from previous versions of the peripheral driver library have been replaced by the EthernetInitExpClk(), EthernetPacketGetNonBlocking(), and EthernetPacketPutNonBlocking() APIs, respectively. Macros have been provided in `ethernet.h` to map the old APIs to the new APIs, allowing existing applications to link and run with the new APIs. It is recommended that new applications utilize the new APIs in favor of the old ones.

8.2.2 Function Documentation

8.2.2.1 EthernetConfigGet

Gets the current configuration of the Ethernet controller.

Prototype:

```
unsigned long
EthernetConfigGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the controller.

Description:

This function will query the control registers of the Ethernet controller and return a bit-mapped configuration value.

See also:

The description of the EthernetConfigSet() function provides detailed information for the bit-mapped configuration values that will be returned.

Returns:

Returns the bit-mapped Ethernet controller configuration value.

8.2.2.2 EthernetConfigSet

Sets the configuration of the Ethernet controller.

Prototype:

```
void
EthernetConfigSet(unsigned long ulBase,
                  unsigned long ulConfig)
```

Parameters:

ulBase is the base address of the controller.

ulConfig is the configuration for the controller.

Description:

After the EthernetInitExpClk() function has been called, this API function can be used to configure the various features of the Ethernet controller.

The Ethernet controller provides three control registers that are used to configure the controller's operation. The transmit control register provides settings to enable full duplex operation, to auto-generate the frame check sequence, and to pad the transmit packets to the minimum length as required by the IEEE standard. The receive control register provides settings to enable reception of packets with bad frame check sequence values and to enable multi-cast or promiscuous modes. The timestamp control register provides settings that enable support logic in the controller that allow the use of the General Purpose Timer 3 to capture timestamps for the transmitted and received packets.

The *ulConfig* parameter is the logical OR of the following values:

- **ETH_CFG_TS_TSEN** - Enable TX and RX interrupt status as CCP timer inputs

- **ETH_CFG_RX_BADCRCDIS** - Disable reception of packets with a bad CRC
- **ETH_CFG_RX_PRMSEN** - Enable promiscuous mode reception (all packets)
- **ETH_CFG_RX_AMULEN** - Enable reception of multicast packets
- **ETH_CFG_TX_DPLXEN** - Enable full duplex transmit mode
- **ETH_CFG_TX_CRCEN** - Enable transmit with auto CRC generation
- **ETH_CFG_TX_PADEN** - Enable padding of transmit data to minimum size

These bit-mapped values are programmed into the transmit, receive, and/or timestamp control register.

Returns:
None.

8.2.2.3 EthernetDisable

Disables the Ethernet controller.

Prototype:
```
void
EthernetDisable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the controller.

Description:
When terminating operations on the Ethernet interface, this function should be called. This function will disable the transmitter and receiver, and will clear out the receive FIFO.

Returns:
None.

8.2.2.4 EthernetEnable

Enables the Ethernet controller for normal operation.

Prototype:
```
void
EthernetEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the controller.

Description:
Once the Ethernet controller has been configured using the EthernetConfigSet() function and the MAC address has been programmed using the EthernetMACAddrSet() function, this API function can be called to enable the controller for normal operation.

This function will enable the controller's transmitter and receiver, and will reset the receive FIFO.

Returns:
None.

8.2.2.5 EthernetInitExpClk

Initializes the Ethernet controller for operation.

Prototype:

```
void
EthernetInitExpClk(unsigned long ulBase,
                   unsigned long ulEthClk)
```

Parameters:

ulBase is the base address of the controller.

ulEthClk is the rate of the clock supplied to the Ethernet module.

Description:

This function will prepare the Ethernet controller for first time use in a given hardware/software configuration. This function should be called before any other Ethernet API functions are called.

The peripheral clock will be the same as the processor clock. This will be the value returned by SysCtlClockGet(), or it can be explicitly hard-coded if it is constant and known (to save the code/execution overhead of a call to SysCtlClockGet()).

This function replaces the original EthernetInit() API and performs the same actions. A macro is provided in `ethernet.h` to map the original API to this API.

Note:

If the device configuration is changed (for example, the system clock is reprogrammed to a different speed), then the Ethernet controller must be disabled by calling the EthernetDisable() function and the controller must be reinitialized by calling the EthernetInitExpClk() function again. After the controller has been reinitialized, the controller should be reconfigured using the appropriate Ethernet API calls.

Returns:

None.

8.2.2.6 EthernetIntClear

Clears Ethernet interrupt sources.

Prototype:

```
void
EthernetIntClear(unsigned long ulBase,
                 unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the controller.

ulIntFlags is a bit mask of the interrupt sources to be cleared.

Description:

The specified Ethernet interrupt sources are cleared so that they no longer assert. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to EthernetIntEnable().

Note:
Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:
None.

8.2.2.7 EthernetIntDisable

Disables individual Ethernet interrupt sources.

Prototype:
```
void
EthernetIntDisable(unsigned long ulBase,
                   unsigned long ulIntFlags)
```

Parameters:
ulBase is the base address of the controller.
ulIntFlags is the bit mask of the interrupt sources to be disabled.

Description:
Disables the indicated Ethernet interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to EthernetIntEnable().

Returns:
None.

8.2.2.8 EthernetIntEnable

Enables individual Ethernet interrupt sources.

Prototype:
```
void
EthernetIntEnable(unsigned long ulBase,
                  unsigned long ulIntFlags)
```

Parameters:
ulBase is the base address of the controller.
ulIntFlags is the bit mask of the interrupt sources to be enabled.

Description:
Enables the indicated Ethernet interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ulIntFlags* parameter is the logical OR of any of the following:

- **ETH_INT_PHY** - An interrupt from the PHY has occurred. The integrated PHY supports a number of interrupt conditions. The PHY register, PHY_MR17, must be read to determine which PHY interrupt has occurred. This register can be read using the EthernetPHYRead() API function.
- **ETH_INT_MDIO** - This interrupt indicates that a transaction on the management interface has completed successfully.
- **ETH_INT_RXER** - This interrupt indicates that an error has occurred during reception of a frame. This error can indicate a length mismatch, a CRC failure, or an error indication from the PHY.
- **ETH_INT_RXOF** - This interrupt indicates that a frame has been received that exceeds the available space in the RX FIFO.
- **ETH_INT_TX** - This interrupt indicates that the packet stored in the TX FIFO has been successfully transmitted.
- **ETH_INT_TXER** - This interrupt indicates that an error has occurred during the transmission of a packet. This error can be either a retry failure during the back-off process, or an invalid length stored in the TX FIFO.
- **ETH_INT_RX** - This interrupt indicates that one (or more) packets are available in the RX FIFO for processing.

Returns:
None.

8.2.2.9 EthernetIntRegister

Registers an interrupt handler for an Ethernet interrupt.

Prototype:
```
void
EthernetIntRegister(unsigned long ulBase,
                    void (*pfnHandler)(void))
```

Parameters:
ulBase is the base address of the controller.
pfnHandler is a pointer to the function to be called when the enabled Ethernet interrupts occur.

Description:
This function sets the handler to be called when the Ethernet interrupt occurs. This will enable the global interrupt in the interrupt controller; specific Ethernet interrupts must be enabled via EthernetIntEnable(). It is the interrupt handler's responsibility to clear the interrupt source.

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

8.2.2.10 EthernetIntStatus

Gets the current Ethernet interrupt status.

Prototype:

```
unsigned long
EthernetIntStatus(unsigned long ulBase,
                  tBoolean bMasked)
```

Parameters:

ulBase is the base address of the controller.

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This returns the interrupt status for the Ethernet controller. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status, enumerated as a bit field of values described in EthernetIntEnable().

8.2.2.11 EthernetIntUnregister

Unregisters an interrupt handler for an Ethernet interrupt.

Prototype:

```
void
EthernetIntUnregister(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the controller.

Description:

This function unregisters the interrupt handler. This will disable the global interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

8.2.2.12 EthernetMACAddrGet

Gets the MAC address of the Ethernet controller.

Prototype:

```
void
EthernetMACAddrGet(unsigned long ulBase,
                   unsigned char *pucMACAddr)
```

Parameters:

ulBase is the base address of the controller.

pucMACAddr is the pointer to the location in which to store the array of MAC-48 address octets.

Description:
This function will read the currently programmed MAC address into the *pucMACAddr* buffer.

See also:
Refer to EthernetMACAddrSet() API description for more details about the MAC address format.

Returns:
None.

8.2.2.13 EthernetMACAddrSet

Sets the MAC address of the Ethernet controller.

Prototype:
```
void
EthernetMACAddrSet(unsigned long ulBase,
                   unsigned char *pucMACAddr)
```

Parameters:
ulBase is the base address of the controller.
pucMACAddr is the pointer to the array of MAC-48 address octets.

Description:
This function will program the IEEE-defined MAC-48 address specified in *pucMACAddr* into the Ethernet controller. This address is used by the Ethernet controller for hardware-level filtering of incoming Ethernet packets (when promiscuous mode is not enabled).

The MAC-48 address is defined as 6 octets, illustrated by the following example address. The numbers are shown in hexadecimal format.

AC-DE-48-00-00-80

In this representation, the first three octets (AC-DE-48) are the Organizationally Unique Identifier (OUI). This is a number assigned by the IEEE to an organization that requests a block of MAC addresses. The last three octets (00-00-80) are a 24-bit number managed by the OUI owner to uniquely identify a piece of hardware within that organization that is to be connected to the Ethernet.

In this representation, the octets are transmitted from left to right, with the "AC" octet being transmitted first and the "80" octet being transmitted last. Within an octet, the bits are transmitted LSB to MSB. For this address, the first bit to be transmitted would be "0", the LSB of "AC", and the last bit to be transmitted would be "1", the MSB of "80".

Returns:
None.

8.2.2.14 EthernetPacketAvail

Check for packet available from the Ethernet controller.

Prototype:
```
tBoolean
EthernetPacketAvail(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the controller.

Description:

The Ethernet controller provides a register that contains the number of packets available in the receive FIFO. When the last bytes of a packet are successfully received (that is, the frame check sequence bytes), the packet count is incremented. Once the packet has been fully read (including the frame check sequence bytes) from the FIFO, the packet count will be decremented.

Returns:

Returns **true** if there are one or more packets available in the receive FIFO, including the current packet being read, and **false** otherwise.

8.2.2.15 EthernetPacketGet

Waits for a packet from the Ethernet controller.

Prototype:

```
long
EthernetPacketGet(unsigned long ulBase,
                  unsigned char *pucBuf,
                  long lBufLen)
```

Parameters:

ulBase is the base address of the controller.

pucBuf is the pointer to the packet buffer.

lBufLen is the maximum number of bytes to be read into the buffer.

Description:

This function reads a packet from the receive FIFO of the controller and places it into *pucBuf*. The function will wait until a packet is available in the FIFO. Then the function will read the entire packet from the receive FIFO. If there are more bytes in the packet than will fit into *pucBuf* (as specified by *lBufLen*), the function will return the negated length of the packet and the buffer will contain *lBufLen* bytes of the packet. Otherwise, the function will return the length of the packet that was read and *pucBuf* will contain the entire packet (excluding the frame check sequence bytes).

Note:

This function is blocking and will not return until a packet arrives.

Returns:

Returns the negated packet length **-n** if the packet is too large for *pucBuf*, and returns the packet length **n** otherwise.

8.2.2.16 EthernetPacketGetNonBlocking

Receives a packet from the Ethernet controller.

Prototype:

```
long
EthernetPacketGetNonBlocking(unsigned long ulBase,
                             unsigned char *pucBuf,
                             long lBufLen)
```

Parameters:

ulBase is the base address of the controller.

pucBuf is the pointer to the packet buffer.

lBufLen is the maximum number of bytes to be read into the buffer.

Description:

This function reads a packet from the receive FIFO of the controller and places it into *pucBuf*. If no packet is available the function will return immediately. Otherwise, the function will read the entire packet from the receive FIFO. If there are more bytes in the packet than will fit into *pucBuf* (as specified by *lBufLen*), the function will return the negated length of the packet and the buffer will contain *lBufLen* bytes of the packet. Otherwise, the function will return the length of the packet that was read and *pucBuf* will contain the entire packet (excluding the frame check sequence bytes).

This function replaces the original EthernetPacketNonBlockingGet() API and performs the same actions. A macro is provided in `ethernet.h` to map the original API to this API.

Note:

This function will return immediately if no packet is available.

Returns:

Returns **0** if no packet is available, the negated packet length **-n** if the packet is too large for *pucBuf*, and the packet length **n** otherwise.

8.2.2.17 EthernetPacketPut

Waits to send a packet from the Ethernet controller.

Prototype:

```
long
EthernetPacketPut(unsigned long ulBase,
                  unsigned char *pucBuf,
                  long lBufLen)
```

Parameters:

ulBase is the base address of the controller.

pucBuf is the pointer to the packet buffer.

lBufLen is number of bytes in the packet to be transmitted.

Description:

This function writes *lBufLen* bytes of the packet contained in *pucBuf* into the transmit FIFO of the controller and then activates the transmitter for this packet. This function will wait until the transmit FIFO is empty. Once space is available, the function will return once *lBufLen* bytes of the packet have been placed into the FIFO and the transmitter has been started. The function will not wait for the transmission to complete. The function will return the negated *lBufLen* if the length is larger than the space available in the transmit FIFO.

Note:
This function blocks and will wait until space is available for the transmit packet before returning.

Returns:
Returns the negated packet length **-lBufLen** if the packet is too large for FIFO, and the packet length **lBufLen** otherwise.

8.2.2.18 EthernetPacketPutNonBlocking

Sends a packet to the Ethernet controller.

Prototype:
```
long
EthernetPacketPutNonBlocking(unsigned long ulBase,
                             unsigned char *pucBuf,
                             long lBufLen)
```

Parameters:
ulBase is the base address of the controller.
pucBuf is the pointer to the packet buffer.
lBufLen is number of bytes in the packet to be transmitted.

Description:
This function writes *lBufLen* bytes of the packet contained in *pucBuf* into the transmit FIFO of the controller and then activates the transmitter for this packet. If no space is available in the FIFO, the function will return immediately. If space is available, the function will return once *lBufLen* bytes of the packet have been placed into the FIFO and the transmitter has been started. The function will not wait for the transmission to complete. The function will return the negated *lBufLen* if the length is larger than the space available in the transmit FIFO.

This function replaces the original EthernetPacketNonBlockingPut() API and performs the same actions. A macro is provided in `ethernet.h` to map the original API to this API.

Note:
This function does not block and will return immediately if no space is available for the transmit packet.

Returns:
Returns **0** if no space is available in the transmit FIFO, the negated packet length **-lBufLen** if the packet is too large for FIFO, and the packet length **lBufLen** otherwise.

8.2.2.19 EthernetPHYRead

Reads from a PHY register.

Prototype:
```
unsigned long
EthernetPHYRead(unsigned long ulBase,
                unsigned char ucRegAddr)
```

Parameters:
ulBase is the base address of the controller.

ucRegAddr is the address of the PHY register to be accessed.

Description:
This function will return the contents of the PHY register specified by *ucRegAddr*.

Returns:
Returns the 16-bit value read from the PHY.

8.2.2.20 EthernetPHYWrite

Writes to the PHY register.

Prototype:
```
void
EthernetPHYWrite(unsigned long ulBase,
                 unsigned char ucRegAddr,
                 unsigned long ulData)
```

Parameters:
ulBase is the base address of the controller.
ucRegAddr is the address of the PHY register to be accessed.
ulData is the data to be written to the PHY register.

Description:
This function will write the *ulData* to the PHY register specified by *ucRegAddr*.

Returns:
None.

8.2.2.21 EthernetSpaceAvail

Checks for packet space available in the Ethernet controller.

Prototype:
```
tBoolean
EthernetSpaceAvail(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the controller.

Description:
The Ethernet controller's transmit FIFO is designed to support a single packet at a time. After the packet has been written into the FIFO, the transmit request bit must be set to enable the transmission of the packet. Only after the packet has been transmitted can a new packet be written into the FIFO. This function will simply check to see if a packet is in progress. If so, there is no space available in the transmit FIFO.

Returns:
Returns **true** if a space is available in the transmit FIFO, and **false** otherwise.

8.3 Programming Example

The following example shows how to use the this API to initialize the Ethernet controller to transmit and receive packets.

```
unsigned char pucMACAddress[6];
unsigned char pucMyRxPacket[];
unsigned char pucMyTxPacket[];
unsigned long ulMyTxPacketLength;

//
// Initialize the Ethernet controller for operation
//
EthernetInitExpClk(ETH_BASE, SysCtlClockGet());

//
// Configure the Ethernet controller for normal operation
// Enable TX Duplex Mode
// Enable TX Padding
//
EthernetConfigSet(ETH_BASE, (ETH_CFG_TX_DPLXEN | ETH_CFG_TX_PADEN));

//
// Program the MAC Address (01-23-45-67-89-AB)
//
pucMACAddress[0] = 0x01;
pucMACAddress[1] = 0x23;
pucMACAddress[2] = 0x45;
pucMACAddress[3] = 0x67;
pucMACAddress[4] = 0x89;
pucMACAddress[5] = 0xAB;
EthernetMACAddrSet(ETH_BASE, pucMACAddress);

//
// Enable the Ethernet controller
//
EthernetEnable(ETH_BASE);

//
// Send a packet.
// (assume that the packet has been filled in appropriately elsewhere
// in the code).
//
EthernetPacketPut(ETH_BASE, pucMyTxPacket, ulMyTxPacketLength);

//
// Wait for a packet to come in.
//
EthernetPacketGet(ETH_BASE, pucMyRxPacket, sizeof(pucMyRxPacket));
```

9 Flash

9.1 Introduction

The flash API provides a set of functions for dealing with the on-chip flash. Functions are provided to program and erase the flash, configure the flash protection, and handle the flash interrupt.

The flash is organized as a set of 1 kB blocks that can be individually erased. Erasing a block causes the entire contents of the block to be reset to all ones. These blocks are paired into a set of 2 kB blocks that can be individually protected. The blocks can be marked as read-only or execute-only, providing differing levels of code protection. Read-only blocks cannot be erased or programmed, protecting the contents of those blocks from being modified. Execute-only blocks cannot be erased or programmed, and can only be read by the processor instruction fetch mechanism, protecting the contents of those blocks from being read by either the processor or by debuggers.

The flash can be programmed on a word-by-word basis. Programming causes 1 bits to become 0 bits (where appropriate); because of this, a word can be repeatedly programmed so long as each programming operation only requires changing 1 bits to 0 bits.

The timing for the flash is automatically handled by the flash controller. In order to do this, the flash controller must know the clock rate of the system in order to be able to time the number of micro-seconds certain signals are asserted. The number of clock cycles per micro-second must be provided to the flash controller for it to accomplish this timing.

The flash controller has the ability to generate an interrupt when an invalid access is attempted (such as reading from execute-only flash). This can be used to validate the operation of a program; the interrupt will keep invalid accesses from being silently ignored, hiding potential bugs. The flash protection can be applied without being permanently enabled; this, along with the interrupt, allows the program to be debugged before the flash protection is permanently applied to the device (which is a non-reversible operation). An interrupt can also be generated when an erase or programming operation has completed.

Depending upon the member of the Stellaris family used, the amount of available flash is 8 KB, 16 KB, 32 KB, 64 KB, 96 KB, 128 KB, or 256 KB.

This driver is contained in `src/flash.c`, with `src/flash.h` containing the API definitions for use by applications.

9.2 API Functions

Functions

- long FlashErase (unsigned long ulAddress)
- void FlashIntClear (unsigned long ulIntFlags)
- void FlashIntDisable (unsigned long ulIntFlags)

- void FlashIntEnable (unsigned long ulIntFlags)
- unsigned long FlashIntGetStatus (tBoolean bMasked)
- void FlashIntRegister (void (*pfnHandler)(void))
- void FlashIntUnregister (void)
- long FlashProgram (unsigned long *pulData, unsigned long ulAddress, unsigned long ulCount)
- tFlashProtection FlashProtectGet (unsigned long ulAddress)
- long FlashProtectSave (void)
- long FlashProtectSet (unsigned long ulAddress, tFlashProtection eProtect)
- unsigned long FlashUsecGet (void)
- void FlashUsecSet (unsigned long ulClocks)
- long FlashUserGet (unsigned long *pulUser0, unsigned long *pulUser1)
- long FlashUserSave (void)
- long FlashUserSet (unsigned long ulUser0, unsigned long ulUser1)

9.2.1 Detailed Description

The flash API is broken into three groups of functions: those that deal with programming the flash, those that deal with flash protection, and those that deal with interrupt handling.

Flash programming is managed with FlashErase(), FlashProgram(), FlashUsecGet(), and FlashUsecSet().

Flash protection is managed with FlashProtectGet(), FlashProtectSet(), and FlashProtectSave().

Interrupt handling is managed with FlashIntRegister(), FlashIntUnregister(), FlashIntEnable(), FlashIntDisable(), FlashIntGetStatus(), and FlashIntClear().

9.2.2 Function Documentation

9.2.2.1 FlashErase

Erases a block of flash.

Prototype:

```
long
FlashErase(unsigned long ulAddress)
```

Parameters:

ulAddress is the start address of the flash block to be erased.

Description:

This function will erase a 1 kB block of the on-chip flash. After erasing, the block will be filled with 0xFF bytes. Read-only and execute-only blocks cannot be erased.

This function will not return until the block has been erased.

Returns:

Returns 0 on success, or -1 if an invalid block address was specified or the block is write-protected.

9.2.2.2 FlashIntClear

Clears flash controller interrupt sources.

Prototype:

```
void
FlashIntClear(unsigned long ulIntFlags)
```

Parameters:

ulIntFlags is the bit mask of the interrupt sources to be cleared. Can be any of the **FLASH_FCMISC_PROGRAM** or **FLASH_FCMISC_AMISC** values.

Description:

The specified flash controller interrupt sources are cleared, so that they no longer assert. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

9.2.2.3 FlashIntDisable

Disables individual flash controller interrupt sources.

Prototype:

```
void
FlashIntDisable(unsigned long ulIntFlags)
```

Parameters:

ulIntFlags is a bit mask of the interrupt sources to be disabled. Can be any of the **FLASH_FCIM_PROGRAM** or **FLASH_FCIM_ACCESS** values.

Description:

Disables the indicated flash controller interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns:

None.

9.2.2.4 FlashIntEnable

Enables individual flash controller interrupt sources.

Prototype:

```
void
FlashIntEnable(unsigned long ulIntFlags)
```

Parameters:

ulIntFlags is a bit mask of the interrupt sources to be enabled. Can be any of the **FLASH_FCIM_PROGRAM** or **FLASH_FCIM_ACCESS** values.

Description:

Enables the indicated flash controller interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns:

None.

9.2.2.5 FlashIntGetStatus

Gets the current interrupt status.

Prototype:

```
unsigned long
FlashIntGetStatus(tBoolean bMasked)
```

Parameters:

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This returns the interrupt status for the flash controller. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

The current interrupt status, enumerated as a bit field of **FLASH_FCMISC_PROGRAM** and **FLASH_FCMISC_AMISC**.

9.2.2.6 FlashIntRegister

Registers an interrupt handler for the flash interrupt.

Prototype:

```
void
FlashIntRegister(void (*pfnHandler)(void))
```

Parameters:

pfnHandler is a pointer to the function to be called when the flash interrupt occurs.

Description:

This sets the handler to be called when the flash interrupt occurs. The flash controller can generate an interrupt when an invalid flash access occurs, such as trying to program or erase a read-only block, or trying to read from an execute-only block. It can also generate an interrupt when a program or erase operation has completed. The interrupt will be automatically enabled when the handler is registered.

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

9.2.2.7 FlashIntUnregister

Unregisters the interrupt handler for the flash interrupt.

Prototype:
```
void
FlashIntUnregister(void)
```

Description:
This function will clear the handler to be called when the flash interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler is no longer called.

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

9.2.2.8 FlashProgram

Programs flash.

Prototype:
```
long
FlashProgram(unsigned long *pulData,
             unsigned long ulAddress,
             unsigned long ulCount)
```

Parameters:
pulData is a pointer to the data to be programmed.
ulAddress is the starting address in flash to be programmed. Must be a multiple of four.
ulCount is the number of bytes to be programmed. Must be a multiple of four.

Description:
This function will program a sequence of words into the on-chip flash. Programming each location consists of the result of an AND operation of the new data and the existing data; in other words bits that contain 1 can remain 1 or be changed to 0, but bits that are 0 cannot be changed to 1. Therefore, a word can be programmed multiple times as long as these rules are followed; if a program operation attempts to change a 0 bit to a 1 bit, that bit will not have its value changed.

Since the flash is programmed one word at a time, the starting address and byte count must both be multiples of four. It is up to the caller to verify the programmed contents, if such verification is required.

This function will not return until the data has been programmed.

Returns:
Returns 0 on success, or -1 if a programming error is encountered.

9.2.2.9 FlashProtectGet

Gets the protection setting for a block of flash.

Prototype:
```
tFlashProtection
FlashProtectGet(unsigned long ulAddress)
```

Parameters:
ulAddress is the start address of the flash block to be queried.

Description:
This function will get the current protection for the specified 2 kB block of flash. Each block can be read/write, read-only, or execute-only. Read/write blocks can be read, executed, erased, and programmed. Read-only blocks can be read and executed. Execute-only blocks can only be executed; processor and debugger data reads are not allowed.

Returns:
Returns the protection setting for this block. See FlashProtectSet() for possible values.

9.2.2.10 FlashProtectSave

Saves the flash protection settings.

Prototype:
```
long
FlashProtectSave(void)
```

Description:
This function will make the currently programmed flash protection settings permanent. This is a non-reversible operation; a chip reset or power cycle will not change the flash protection.

This function will not return until the protection has been saved.

Returns:
Returns 0 on success, or -1 if a hardware error is encountered.

9.2.2.11 FlashProtectSet

Sets the protection setting for a block of flash.

Prototype:
```
long
FlashProtectSet(unsigned long ulAddress,
                tFlashProtection eProtect)
```

Parameters:

ulAddress is the start address of the flash block to be protected.

eProtect is the protection to be applied to the block. Can be one of **FlashReadWrite**, **FlashReadOnly**, or **FlashExecuteOnly**.

Description:

This function will set the protection for the specified 2 kB block of flash. Blocks which are read/write can be made read-only or execute-only. Blocks which are read-only can be made execute-only. Blocks which are execute-only cannot have their protection modified. Attempts to make the block protection less stringent (that is, read-only to read/write) will result in a failure (and be prevented by the hardware).

Changes to the flash protection are maintained only until the next reset. This allows the application to be executed in the desired flash protection environment to check for inappropriate flash access (via the flash interrupt). To make the flash protection permanent, use the FlashProtectSave() function.

Returns:

Returns 0 on success, or -1 if an invalid address or an invalid protection was specified.

9.2.2.12 FlashUsecGet

Gets the number of processor clocks per micro-second.

Prototype:

```
unsigned long
FlashUsecGet(void)
```

Description:

This function returns the number of clocks per micro-second, as presently known by the flash controller.

Returns:

Returns the number of processor clocks per micro-second.

9.2.2.13 FlashUsecSet

Sets the number of processor clocks per micro-second.

Prototype:

```
void
FlashUsecSet(unsigned long ulClocks)
```

Parameters:

ulClocks is the number of processor clocks per micro-second.

Description:

This function is used to tell the flash controller the number of processor clocks per micro-second. This value must be programmed correctly or the flash most likely will not program correctly; it has no affect on reading flash.

Returns:

None.

9.2.2.14 FlashUserGet

Gets the user registers.

Prototype:

```
long
FlashUserGet(unsigned long *pulUser0,
             unsigned long *pulUser1)
```

Parameters:

pulUser0 is a pointer to the location to store USER Register 0.

pulUser1 is a pointer to the location to store USER Register 1.

Description:

This function will read the contents of user registers (0 and 1), and store them in the specified locations.

Returns:

Returns 0 on success, or -1 if a hardware error is encountered.

9.2.2.15 FlashUserSave

Saves the user registers.

Prototype:

```
long
FlashUserSave(void)
```

Description:

This function will make the currently programmed user register settings permanent. This is a non-reversible operation; a chip reset or power cycle will not change this setting.

This function will not return until the protection has been saved.

Returns:

Returns 0 on success, or -1 if a hardware error is encountered.

9.2.2.16 FlashUserSet

Sets the user registers.

Prototype:

```
long
FlashUserSet(unsigned long ulUser0,
             unsigned long ulUser1)
```

Parameters:

ulUser0 is the value to store in USER Register 0.

ulUser1 is the value to store in USER Register 1.

Description:

This function will set the contents of the user registers (0 and 1) to the specified values.

Returns:

Returns 0 on success, or -1 if a hardware error is encountered.

9.3 Programming Example

The following example shows how to use the flash API to erase a block of the flash and program a few words.

```
unsigned long pulData[2];

//
// Set the uSec value to 20, indicating that the processor is running at
// 20 MHz.
//
FlashUsecSet(20);

//
// Erase a block of the flash.
//
FlashErase(0x800);

//
// Program some data into the newly erased block of the flash.
//
pulData[0] = 0x12345678;
pulData[1] = 0x56789abc;
FlashProgram(pulData, 0x800, sizeof(pulData));
```

10 GPIO

10.1 Introduction

The GPIO module provides control for up to eight independent GPIO pins (the actual number present depend upon the GPIO port and part number). Each pin has the following capabilities:

- Can be configured as an input or an output. On reset, they default to being an input.
- In input mode, can generate interrupts on high level, low level, rising edge, falling edge, or both edges.
- In output mode, can be configured for 2 mA, 4 mA, or 8 mA drive strength. The 8 mA drive strength configuration has optional slew rate control to limit the rise and fall times of the signal. On reset, they default to 2 mA drive strength.
- Optional weak pull-up or pull-down resistors. On reset, they default to a weak pull-up.
- Optional open-drain operation. On reset, they default to standard push/pull operation.
- Can be configured to be a GPIO or a peripheral pin. On reset, they default to being GPIOs. Note that not all pins on all parts have peripheral functions, in which case the pin is only useful as a GPIO (that is, when configured for peripheral function the pin will not do anything useful).

Most of the GPIO functions can operate on more than one GPIO pin (within a single module) at a time. The *ucPins* parameter to these functions is used to specify the pins that are affected; the GPIO pins whose corresponding bits in this parameter that are set will be affected (where pin 0 is in bit 0, pin 1 in bit 1, and so on). For example, if *ucPins* is 0x09, then pins 0 and 3 will be affected by the function.

This is most useful for the GPIOPinRead() and GPIOPinWrite() functions; a read will return only the value of the requested pins (with the other pin values masked out) and a write will affect the requested pins simultaneously (that is, the state of multiple GPIO pins can be changed at the same time). This data masking for the GPIO pin state occurs in the hardware; a single read or write is issued to the hardware, which interprets some of the address bits as an indication of the GPIO pins to operate upon (and therefore the ones to not affect). See the part data sheet for details of the GPIO data register address-based bit masking.

For functions that have a *ucPin* (singular) parameter, only a single pin is affected by the function. In this case, this value specifies the pin number (that is, 0 through 7).

This driver is contained in `src/gpio.c`, with `src/gpio.h` containing the API definitions for use by applications.

10.2 API Functions

Functions

- unsigned long GPIODirModeGet (unsigned long ulPort, unsigned char ucPin)

- void GPIODirModeSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulPinIO)
- unsigned long GPIOIntTypeGet (unsigned long ulPort, unsigned char ucPin)
- void GPIOIntTypeSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulIntType)
- void GPIOPadConfigGet (unsigned long ulPort, unsigned char ucPin, unsigned long *pulStrength, unsigned long *pulPinType)
- void GPIOPadConfigSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulStrength, unsigned long ulPinType)
- void GPIOPinIntClear (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinIntDisable (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinIntEnable (unsigned long ulPort, unsigned char ucPins)
- long GPIOPinIntStatus (unsigned long ulPort, tBoolean bMasked)
- long GPIOPinRead (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeADC (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeCAN (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeComparator (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeGPIOInput (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeGPIOOutput (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeGPIOOutputOD (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeI2C (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypePWM (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeQEI (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeSSI (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeTimer (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeUART (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinTypeUSBDigital (unsigned long ulPort, unsigned char ucPins)
- void GPIOPinWrite (unsigned long ulPort, unsigned char ucPins, unsigned char ucVal)
- void GPIOPortIntRegister (unsigned long ulPort, void (*pfnIntHandler)(void))
- void GPIOPortIntUnregister (unsigned long ulPort)

10.2.1 Detailed Description

The GPIO API is broken into three groups of functions: those that deal with configuring the GPIO pins, those that deal with interrupts, and those that access the pin value.

The GPIO pins are configured with GPIODirModeSet() and GPIOPadConfigSet(). The configuration can be read back with GPIODirModeGet() and GPIOPadConfigGet(). There are also convenience functions for configuring the pin in the required or recommended configuration for a particular peripheral; these are GPIOPinTypeCAN(), GPIOPinTypeComparator(), GPIOPinTypeGPIOInput(), GPIOPinTypeGPIOOutput(), GPIOPinTypeGPIOOutputOD(), GPIOPinTypeI2C(), GPIOPinTypePWM(), GPIOPinTypeQEI(), GPIOPinTypeSSI(), GPIOPinTypeTimer(), and GPIOPinTypeUART().

The GPIO interrupts are handled with GPIOIntTypeSet(), GPIOIntTypeGet(), GPIOPinIntEnable(), GPIOPinIntDisable(), GPIOPinIntStatus(), GPIOPinIntClear(), GPIOPortIntRegister(), and GPIOPortIntUnregister().

The GPIO pin state is accessed with GPIOPinRead() and GPIOPinWrite().

10.2.2 Function Documentation

10.2.2.1 GPIODirModeGet

Gets the direction and mode of a pin.

Prototype:

```
unsigned long
GPIODirModeGet(unsigned long ulPort,
               unsigned char ucPin)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPin is the pin number.

Description:

This function gets the direction and control mode for a specified pin on the selected GPIO port. The pin can be configured as either an input or output under software control, or it can be under hardware control. The type of control and direction are returned as an enumerated data type.

Returns:

Returns one of the enumerated data types described for GPIODirModeSet().

10.2.2.2 GPIODirModeSet

Sets the direction and mode of the specified pin(s).

Prototype:

```
void
GPIODirModeSet(unsigned long ulPort,
               unsigned char ucPins,
               unsigned long ulPinIO)
```

Parameters:

ulPort is the base address of the GPIO port

ucPins is the bit-packed representation of the pin(s).

ulPinIO is the pin direction and/or mode.

Description:

This function will set the specified pin(s) on the selected GPIO port as either an input or output under software control, or it will set the pin to be under hardware control.

The parameter *ulPinIO* is an enumerated data type that can be one of the following values:

- **GPIO_DIR_MODE_IN**
- **GPIO_DIR_MODE_OUT**
- **GPIO_DIR_MODE_HW**

where **GPIO_DIR_MODE_IN** specifies that the pin will be programmed as a software controlled input, **GPIO_DIR_MODE_OUT** specifies that the pin will be programmed as a software controlled output, and **GPIO_DIR_MODE_HW** specifies that the pin will be placed under hardware control.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:
None.

10.2.2.3 GPIOIntTypeGet

Gets the interrupt type for a pin.

Prototype:
```
unsigned long
GPIOIntTypeGet(unsigned long ulPort,
               unsigned char ucPin)
```

Parameters:
ulPort is the base address of the GPIO port.
ucPin is the pin number.

Description:
This function gets the interrupt type for a specified pin on the selected GPIO port. The pin can be configured as a falling edge, rising edge, or both edge detected interrupt, or it can be configured as a low level or high level detected interrupt. The type of interrupt detection mechanism is returned as an enumerated data type.

Returns:
Returns one of the enumerated data types described for GPIOIntTypeSet().

10.2.2.4 GPIOIntTypeSet

Sets the interrupt type for the specified pin(s).

Prototype:
```
void
GPIOIntTypeSet(unsigned long ulPort,
               unsigned char ucPins,
               unsigned long ulIntType)
```

Parameters:
ulPort is the base address of the GPIO port.
ucPins is the bit-packed representation of the pin(s).
ulIntType specifies the type of interrupt trigger mechanism.

Description:
This function sets up the various interrupt trigger mechanisms for the specified pin(s) on the selected GPIO port.

The parameter *ulIntType* is an enumerated data type that can be one of the following values:

- **GPIO_FALLING_EDGE**

- **GPIO_RISING_EDGE**
- **GPIO_BOTH_EDGES**
- **GPIO_LOW_LEVEL**
- **GPIO_HIGH_LEVEL**

where the different values describe the interrupt detection mechanism (edge or level) and the particular triggering event (falling, rising, or both edges for edge detect, low or high for level detect).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:
In order to avoid any spurious interrupts, the user must ensure that the GPIO inputs remain stable for the duration of this function.

Returns:
None.

10.2.2.5 GPIOPadConfigGet

Gets the pad configuration for a pin.

Prototype:
```
void
GPIOPadConfigGet(unsigned long ulPort,
                 unsigned char ucPin,
                 unsigned long *pulStrength,
                 unsigned long *pulPinType)
```

Parameters:
ulPort is the base address of the GPIO port.
ucPin is the pin number.
pulStrength is a pointer to storage for the output drive strength.
pulPinType is a pointer to storage for the output drive type.

Description:
This function gets the pad configuration for a specified pin on the selected GPIO port. The values returned in *pulStrength* and *pulPinType* correspond to the values used in GPIOPadConfigSet(). This function also works for pin(s) configured as input pin(s); however, the only meaningful data returned is whether the pin is terminated with a pull-up or down resistor.

Returns:
None

10.2.2.6 GPIOPadConfigSet

Sets the pad configuration for the specified pin(s).

Prototype:

```
void
GPIOPadConfigSet(unsigned long ulPort,
                 unsigned char ucPins,
                 unsigned long ulStrength,
                 unsigned long ulPinType)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

ulStrength specifies the output drive strength.

ulPinType specifies the pin type.

Description:

This function sets the drive strength and type for the specified pin(s) on the selected GPIO port. For pin(s) configured as input ports, the pad is configured as requested, but the only real effect on the input is the configuration of the pull-up or pull-down termination.

The parameter *ulStrength* can be one of the following values:

- **GPIO_STRENGTH_2MA**
- **GPIO_STRENGTH_4MA**
- **GPIO_STRENGTH_8MA**
- **GPIO_STRENGTH_8MA_SC**

where **GPIO_STRENGTH_xMA** specifies either 2, 4, or 8 mA output drive strength, and **GPIO_OUT_STRENGTH_8MA_SC** specifies 8 mA output drive with slew control.

The parameter *ulPinType* can be one of the following values:

- **GPIO_PIN_TYPE_STD**
- **GPIO_PIN_TYPE_STD_WPU**
- **GPIO_PIN_TYPE_STD_WPD**
- **GPIO_PIN_TYPE_OD**
- **GPIO_PIN_TYPE_OD_WPU**
- **GPIO_PIN_TYPE_OD_WPD**
- **GPIO_PIN_TYPE_ANALOG**

where **GPIO_PIN_TYPE_STD**∗ specifies a push-pull pin, **GPIO_PIN_TYPE_OD**∗ specifies an open-drain pin, ∗**_WPU** specifies a weak pull-up, ∗**_WPD** specifies a weak pull-down, and **GPIO_PIN_TYPE_ANALOG** specifies an analog input (for the comparators).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

10.2.2.7 GPIOPinIntClear

Clears the interrupt for the specified pin(s).

Prototype:

```
void
GPIOPinIntClear(unsigned long ulPort,
                unsigned char ucPins)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

Description:

Clears the interrupt for the specified pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

10.2.2.8 GPIOPinIntDisable

Disables interrupts for the specified pin(s).

Prototype:

```
void
GPIOPinIntDisable(unsigned long ulPort,
                  unsigned char ucPins)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

Description:

Masks the interrupt for the specified pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

10.2.2.9 GPIOPinIntEnable

Enables interrupts for the specified pin(s).

Prototype:

```
void
GPIOPinIntEnable(unsigned long ulPort,
                 unsigned char ucPins)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

Description:

Unmasks the interrupt for the specified pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

10.2.2.10 GPIOPinIntStatus

Gets interrupt status for the specified GPIO port.

Prototype:

```
long
GPIOPinIntStatus(unsigned long ulPort,
                 tBoolean bMasked)
```

Parameters:

ulPort is the base address of the GPIO port.

bMasked specifies whether masked or raw interrupt status is returned.

Description:

If *bMasked* is set as **true**, then the masked interrupt status is returned; otherwise, the raw interrupt status will be returned.

Returns:

Returns a bit-packed byte, where each bit that is set identifies an active masked or raw interrupt, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on. Bits 31:8 should be ignored.

10.2.2.11 GPIOPinRead

Reads the values present of the specified pin(s).

Prototype:

```
long
GPIOPinRead(unsigned long ulPort,
            unsigned char ucPins)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

Description:

The values at the specified pin(s) are read, as specified by *ucPins*. Values are returned for both input and output pin(s), and the value for pin(s) that are not specified by *ucPins* are set to 0.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

Returns a bit-packed byte providing the state of the specified pin, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on. Any bit that is not specified by *ucPins* is returned as a 0. Bits 31:8 should be ignored.

10.2.2.12 GPIOPinTypeADC

Configures pin(s) for use as analog-to-digital converter inputs.

Prototype:

```
void
GPIOPinTypeADC(unsigned long ulPort,
               unsigned char ucPins)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

Description:

The analog-to-digital converter input pins must be properly configured to function correctly on DustDevil-class devices. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This cannot be used to turn any pin into an ADC input; it only configures an ADC input pin for proper operation.

Returns:

None.

10.2.2.13 GPIOPinTypeCAN

Configures pin(s) for use as a CAN device.

Prototype:

```
void
GPIOPinTypeCAN(unsigned long ulPort,
               unsigned char ucPins)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

Description:

The CAN pins must be properly configured for the CAN peripherals to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This cannot be used to turn any pin into a CAN pin; it only configures a CAN pin for proper operation.

Returns:

None.

10.2.2.14 GPIOPinTypeComparator

Configures pin(s) for use as an analog comparator input.

Prototype:

```
void
GPIOPinTypeComparator(unsigned long ulPort,
                      unsigned char ucPins)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

Description:

The analog comparator input pins must be properly configured for the analog comparator to function correctly. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This cannot be used to turn any pin into an analog comparator input; it only configures an analog comparator pin for proper operation.

Returns:
None.

10.2.2.15 GPIOPinTypeGPIOInput

Configures pin(s) for use as GPIO inputs.

Prototype:
```
void
GPIOPinTypeGPIOInput(unsigned long ulPort,
                     unsigned char ucPins)
```

Parameters:
ulPort is the base address of the GPIO port.
ucPins is the bit-packed representation of the pin(s).

Description:
The GPIO pins must be properly configured in order to function correctly as GPIO inputs; this is especially true of Fury-class devices where the digital input enable is turned off by default. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:
None.

10.2.2.16 GPIOPinTypeGPIOOutput

Configures pin(s) for use as GPIO outputs.

Prototype:
```
void
GPIOPinTypeGPIOOutput(unsigned long ulPort,
                      unsigned char ucPins)
```

Parameters:
ulPort is the base address of the GPIO port.
ucPins is the bit-packed representation of the pin(s).

Description:
The GPIO pins must be properly configured in order to function correctly as GPIO outputs; this is especially true of Fury-class devices where the digital input enable is turned off by default. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:
None.

10.2.2.17 GPIOPinTypeGPIOOutputOD

Configures pin(s) for use as GPIO open drain outputs.

Prototype:

```
void
GPIOPinTypeGPIOOutputOD(unsigned long ulPort,
                        unsigned char ucPins)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

Description:

The GPIO pins must be properly configured in order to function correctly as GPIO outputs; this is especially true of Fury-class devices where the digital input enable is turned off by default. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

10.2.2.18 GPIOPinTypeI2C

Configures pin(s) for use by the I2C peripheral.

Prototype:

```
void
GPIOPinTypeI2C(unsigned long ulPort,
               unsigned char ucPins)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

Description:

The I2C pins must be properly configured for the I2C peripheral to function correctly. This function provides the proper configuration for those pin(s).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This cannot be used to turn any pin into an I2C pin; it only configures an I2C pin for proper operation.

Returns:

None.

10.2.2.19 GPIOPinTypePWM

Configures pin(s) for use by the PWM peripheral.

Prototype:

```
void
GPIOPinTypePWM(unsigned long ulPort,
               unsigned char ucPins)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

Description:

The PWM pins must be properly configured for the PWM peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This cannot be used to turn any pin into a PWM pin; it only configures a PWM pin for proper operation.

Returns:

None.

10.2.2.20 GPIOPinTypeQEI

Configures pin(s) for use by the QEI peripheral.

Prototype:

```
void
GPIOPinTypeQEI(unsigned long ulPort,
               unsigned char ucPins)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

Description:

The QEI pins must be properly configured for the QEI peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, not using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This cannot be used to turn any pin into a QEI pin; it only configures a QEI pin for proper operation.

Returns:
None.

10.2.2.21 GPIOPinTypeSSI

Configures pin(s) for use by the SSI peripheral.

Prototype:
```
void
GPIOPinTypeSSI(unsigned long ulPort,
               unsigned char ucPins)
```

Parameters:
ulPort is the base address of the GPIO port.
ucPins is the bit-packed representation of the pin(s).

Description:
The SSI pins must be properly configured for the SSI peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:
This cannot be used to turn any pin into a SSI pin; it only configures a SSI pin for proper operation.

Returns:
None.

10.2.2.22 GPIOPinTypeTimer

Configures pin(s) for use by the Timer peripheral.

Prototype:
```
void
GPIOPinTypeTimer(unsigned long ulPort,
                 unsigned char ucPins)
```

Parameters:
ulPort is the base address of the GPIO port.
ucPins is the bit-packed representation of the pin(s).

Description:
The CCP pins must be properly configured for the timer peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:
This cannot be used to turn any pin into a timer pin; it only configures a timer pin for proper operation.

Returns:
None.

10.2.2.23 GPIOPinTypeUART

Configures pin(s) for use by the UART peripheral.

Prototype:
```
void
GPIOPinTypeUART(unsigned long ulPort,
                unsigned char ucPins)
```

Parameters:
ulPort is the base address of the GPIO port.
ucPins is the bit-packed representation of the pin(s).

Description:
The UART pins must be properly configured for the UART peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:
This cannot be used to turn any pin into a UART pin; it only configures a UART pin for proper operation.

Returns:
None.

10.2.2.24 GPIOPinTypeUSBDigital

Configures pin(s) for use by the USB peripheral.

Prototype:
```
void
GPIOPinTypeUSBDigital(unsigned long ulPort,
                      unsigned char ucPins)
```

Parameters:
ulPort is the base address of the USB port.
ucPins is the bit-packed representation of the pin(s).

Description:

Some USB pins must be properly configured for the USB peripheral to function correctly. This function provides a typical configuration for the digital USB pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

This cannot be used to turn any pin into a USB pin; it only configures a USB pin for proper operation.

Returns:

None.

10.2.2.25 GPIOPinWrite

Writes a value to the specified pin(s).

Prototype:

```
void
GPIOPinWrite(unsigned long ulPort,
             unsigned char ucPins,
             unsigned char ucVal)
```

Parameters:

ulPort is the base address of the GPIO port.

ucPins is the bit-packed representation of the pin(s).

ucVal is the value to write to the pin(s).

Description:

Writes the corresponding bit values to the output pin(s) specified by *ucPins*. Writing to a pin configured as an input pin has no effect.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

10.2.2.26 GPIOPortIntRegister

Registers an interrupt handler for a GPIO port.

Prototype:

```
void
GPIOPortIntRegister(unsigned long ulPort,
                    void (*pfnIntHandler)(void))
```

Parameters:

ulPort is the base address of the GPIO port.

pfnIntHandler is a pointer to the GPIO port interrupt handling function.

Description:

This function will ensure that the interrupt handler specified by *pfnIntHandler* is called when an interrupt is detected from the selected GPIO port. This function will also enable the corresponding GPIO interrupt in the interrupt controller; individual pin interrupts and interrupt sources must be enabled with GPIOPinIntEnable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

10.2.2.27 GPIOPortIntUnregister

Removes an interrupt handler for a GPIO port.

Prototype:

```
void
GPIOPortIntUnregister(unsigned long ulPort)
```

Parameters:

ulPort is the base address of the GPIO port.

Description:

This function will unregister the interrupt handler for the specified GPIO port. This function will also disable the corresponding GPIO port interrupt in the interrupt controller; individual GPIO interrupts and interrupt sources must be disabled with GPIOPinIntDisable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

10.3 Programming Example

The following example shows how to use the GPIO API to initialize the GPIO, enable interrupts, read data from pins, and write data to pins.

```
int iVal;

//
// Register the port-level interrupt handler. This handler is the
// first level interrupt handler for all the pin interrupts.
//
GPIOPortIntRegister(GPIO_PORTA_BASE, PortAIntHandler);
```

```
//
// Initialize the GPIO pin configuration.
//
// Set pins 2, 4, and 5 as input, SW controlled.
//
GPIOPinTypeGPIOInput(GPIO_PORTA_BASE,
                     GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5);

//
// Set pins 0 and 3 as output, SW controlled.
//
GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_3);

//
// Make pins 2 and 4 rising edge triggered interrupts.
//
GPIOIntTypeSet(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_4, GPIO_RISING_EDGE);

//
// Make pin 5 high level triggered interrupts.
//
GPIOIntTypeSet(GPIO_PORTA_BASE, GPIO_PIN_5, GPIO_HIGH_LEVEL);

//
// Read some pins.
//
iVal = GPIOPinRead(GPIO_PORTA_BASE,
                   (GPIO_PIN_0 | GPIO_PIN_2 | GPIO_PIN_3 |
                    GPIO_PIN_4 | GPIO_PIN_5));

//
// Write some pins.  Even though pins 2, 4, and 5 are specified, those
// pins are unaffected by this write since they are configured as inputs.
// At the end of this write, pin 0 will be a 0, and pin 3 will be a 1.
//
GPIOPinWrite(GPIO_PORTA_BASE,
             (GPIO_PIN_0 | GPIO_PIN_2 | GPIO_PIN_3 |
              GPIO_PIN_4 | GPIO_PIN_5),
             0xF4);

//
// Enable the pin interrupts.
//
GPIOPinIntEnable(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5);
```

11 Hibernation Module

11.1 Introduction

The Hibernate API provides a set of functions for using the Hibernation module on the Stellaris microcontroller. The Hibernation module allows the software application to cause power to be removed from the microcontroller, and then be powered on later based on specific time or a signal on the external **WAKE** pin. The API provides functions to configure wake conditions, manage interrupts, read status, save and restore program state information, and request hibernation mode.

Some of the features of the Hibernation module are:

- 32-bit real time clock
- Trim register for fine tuning the RTC rate
- Two RTC match registers for generating RTC events
- External **WAKE** pin to initiate a wake-up
- Low-battery detection
- 64 32-bit words of non-volatile memory
- Programmable interrupts for hibernation events

This driver is contained in `src/hibernate.c`, with `src/hibernate.h` containing the API definitions for use by applications.

11.2 API Functions

Functions

- void HibernateClockSelect (unsigned long ulClockInput)
- void HibernateDataGet (unsigned long *pulData, unsigned long ulCount)
- void HibernateDataSet (unsigned long *pulData, unsigned long ulCount)
- void HibernateDisable (void)
- void HibernateEnableExpClk (unsigned long ulHibClk)
- void HibernateIntClear (unsigned long ulIntFlags)
- void HibernateIntDisable (unsigned long ulIntFlags)
- void HibernateIntEnable (unsigned long ulIntFlags)
- void HibernateIntRegister (void (*pfnHandler)(void))
- unsigned long HibernateIntStatus (tBoolean bMasked)
- void HibernateIntUnregister (void)
- unsigned int HibernateIsActive (void)
- unsigned long HibernateLowBatGet (void)

- void HibernateLowBatSet (unsigned long ulLowBatFlags)
- void HibernateRequest (void)
- void HibernateRTCDisable (void)
- void HibernateRTCEnable (void)
- unsigned long HibernateRTCGet (void)
- unsigned long HibernateRTCMatch0Get (void)
- void HibernateRTCMatch0Set (unsigned long ulMatch)
- unsigned long HibernateRTCMatch1Get (void)
- void HibernateRTCMatch1Set (unsigned long ulMatch)
- void HibernateRTCSet (unsigned long ulRTCValue)
- unsigned long HibernateRTCTrimGet (void)
- void HibernateRTCTrimSet (unsigned long ulTrim)
- unsigned long HibernateWakeGet (void)
- void HibernateWakeSet (unsigned long ulWakeFlags)

11.2.1 Detailed Description

The Hibernation module must be enabled before it can be used. Use the HibernateEnableExpClk() function to enable it. If a crystal is used for the clock source, then the initializing code must allow time for the crystal to stabilize after calling the HibernateEnableExpClk() function. Refer to the device data sheet for information about crystal stabilization time. If an oscillator is used, then no delay is necessary. After the module is enabled, the clock source must be configured by calling HibernateClockSelect().

In order to use the RTC feature of the Hibernation module, the RTC must be enabled by calling HibernateRTCEnable(). It can be later disabled by calling HibernateRTCDisable(). These functions can be called at any time to start and stop the RTC. The RTC value can be read or set by using the HibernateRTCGet() and HibernateRTCSet() functions. The two match registers can be read and set by using the HibernateRTCMatch0Get(), HibernateRTCMatch0Set(), HibernateRTCMatch1Get(), and HibernateRTCMatch1Set() functions. The real-time clock rate can be adjusted by using the trim register. Use the HibernateRTCTrimGet() and HibernateRTCTrimSet() functions for this purpose.

Application state information can be stored in the non-volatile memory of the Hibernation module when the processor is powered off. Use the HibernateDataSet() and HibernateDataGet() functions to access the non-volatile memory area.

The module can be configured to wake when the external **WAKE** pin is asserted, or when an RTC match occurs, or both. Use the HibernateWakeSet() function to configure the wake conditions. The present configuration can be read by calling HibernateWakeGet().

The Hibernation module can detect a low battery and signal the processor. It can also be configured to abort a hibernation request if the battery voltage is too low. Use the HibernateLowBatSet() and HibernateLowBatGet() functions to configure this feature.

Several functions are provided for managing interrupts. Use the HibernateIntRegister() and HibernateIntUnregister() functions to install or uninstall an interrupt handler into the vector table. Refer to the IntRegister() function for notes about using the interrupt vector table. The module can generate several different interrupts. Use the HibernateIntEnable() and HibernateIntDisable() functions to enable and disable specific interrupt sources. The present interrupt status can be found by calling HibernateIntStatus(). In the interrupt handler, all pending interrupts must be cleared. Use the HibernateIntClear() function to clear pending interrupts.

Finally, once the module is appropriately configured, the state saved, and the software application is ready to hibernate, call the HibernateRequest() function. This will initiate the sequence to remove power from the processor. At a power-on reset, the software application can use the HibernateIsActive() function to determine if the Hibernation module is already active and therefore does not need to be enabled. This can provide a hint to the software that the processor is waking from hibernation instead of a cold start. The software can then use the HibernateIntStatus() and HibernateDataGet() functions to discover the cause of the wake and to get the saved system state.

The HibernateEnable() API from previous versions of the peripheral driver library has been replaced by the HibernateEnableExpClk() API. A macro has been provided in `hibernate.h` to map the old API to the new API, allowing existing applications to link and run with the new API. It is recommended that new applications utilize the new API in favor of the old one.

11.2.2 Function Documentation

11.2.2.1 HibernateClockSelect

Selects the clock input for the Hibernation module.

Prototype:

```
void
HibernateClockSelect(unsigned long ulClockInput)
```

Parameters:

ulClockInput specifies the clock input.

Description:

Configures the clock input for the Hibernation module. The configuration option chosen depends entirely on hardware design. The clock input for the module will either be a 32.768 kHz oscillator or a 4.194304 MHz crystal. The *ulClockFlags* parameter must be one of the following:

- **HIBERNATE_CLOCK_SEL_RAW** - use the raw signal from a 32.768 kHz oscillator.
- **HIBERNATE_CLOCK_SEL_DIV128** - use the crystal input, divided by 128.

Returns:

None.

11.2.2.2 HibernateDataGet

Reads a set of data from the non-volatile memory of the Hibernation module.

Prototype:

```
void
HibernateDataGet(unsigned long *pulData,
                 unsigned long ulCount)
```

Parameters:

pulData points to a location where the data that is read from the Hibernation module will be stored.

ulCount is the count of 32-bit words to read.

Description:
Retrieves a set of data from the Hibernation module non-volatile memory that was previously stored with the HibernateDataSet() function. The caller must ensure that *pulData* points to a large enough memory block to hold all the data that is read from the non-volatile memory.

Returns:
None.

11.2.2.3 HibernateDataSet

Stores data in the non-volatile memory of the Hibernation module.

Prototype:
```
void
HibernateDataSet(unsigned long *pulData,
                 unsigned long ulCount)
```

Parameters:
pulData points to the data that the caller wants to store in the memory of the Hibernation module.
ulCount is the count of 32-bit words to store.

Description:
Stores a set of data in the Hibernation module non-volatile memory. This memory will be preserved when the power to the processor is turned off, and can be used to store application state information which will be available when the processor wakes. Up to 64 32-bit words can be stored in the non-volatile memory. The data can be restored by calling the HibernateDataGet() function.

Returns:
None.

11.2.2.4 HibernateDisable

Disables the Hibernation module for operation.

Prototype:
```
void
HibernateDisable(void)
```

Description:
Disables the Hibernation module for operation. After this function is called, none of the Hibernation module features are available.

Returns:
None.

11.2.2.5 HibernateEnableExpClk

Enables the Hibernation module for operation.

Prototype:

```
void
HibernateEnableExpClk(unsigned long ulHibClk)
```

Parameters:

ulHibClk is the rate of the clock supplied to the Hibernation module.

Description:

Enables the Hibernation module for operation. This function should be called before any of the Hibernation module features are used.

The peripheral clock will be the same as the processor clock. This will be the value returned by SysCtlClockGet(), or it can be explicitly hard-coded if it is constant and known (to save the code/execution overhead of a call to SysCtlClockGet()).

This function replaces the original HibernateEnable() API and performs the same actions. A macro is provided in `hibernate.h` to map the original API to this API.

Returns:

None.

11.2.2.6 HibernateIntClear

Clears pending interrupts from the Hibernation module.

Prototype:

```
void
HibernateIntClear(unsigned long ulIntFlags)
```

Parameters:

ulIntFlags is the bit mask of the interrupts to be cleared.

Description:

Clears the specified interrupt sources. This must be done from within the interrupt handler or else the handler will be called again upon exit.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to the HibernateIntEnable() function.

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

11.2.2.7 HibernateIntDisable

Disables interrupts for the Hibernation module.

Prototype:

```
void
HibernateIntDisable(unsigned long ulIntFlags)
```

Parameters:

ulIntFlags is the bit mask of the interrupts to be disabled.

Description:

Disables the specified interrupt sources from the Hibernation module.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to the HibernateIntEnable() function.

Returns:

None.

11.2.2.8 HibernateIntEnable

Enables interrupts for the Hibernation module.

Prototype:

```
void
HibernateIntEnable(unsigned long ulIntFlags)
```

Parameters:

ulIntFlags is the bit mask of the interrupts to be enabled.

Description:

Enables the specified interrupt sources from the Hibernation module.

The *ulIntFlags* parameter must be the logical OR of any combination of the following:

- **HIBERNATE_INT_PIN_WAKE** - wake from pin interrupt
- **HIBERNATE_INT_LOW_BAT** - low battery interrupt
- **HIBERNATE_INT_RTC_MATCH_0** - RTC match 0 interrupt
- **HIBERNATE_INT_RTC_MATCH_1** - RTC match 1 interrupt

Returns:

None.

11.2.2.9 HibernateIntRegister

Registers an interrupt handler for the Hibernation module interrupt.

Prototype:

```
void
HibernateIntRegister(void (*pfnHandler)(void))
```

Parameters:

pfnHandler points to the function to be called when a hibernation interrupt occurs.

Description:

Registers the interrupt handler in the system interrupt controller. The interrupt is enabled at the global level, but individual interrupt sources must still be enabled with a call to HibernateIntEnable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

11.2.2.10 HibernateIntStatus

Gets the current interrupt status of the Hibernation module.

Prototype:

```
unsigned long
HibernateIntStatus(tBoolean bMasked)
```

Parameters:

bMasked is false to retrieve the raw interrupt status, and true to retrieve the masked interrupt status.

Description:

Returns the interrupt status of the Hibernation module. The caller can use this to determine the cause of a hibernation interrupt. Either the masked or raw interrupt status can be returned.

Returns:

Returns the interrupt status as a bit field with the values as described in the HibernateIntEnable() function.

11.2.2.11 HibernateIntUnregister

Unregisters an interrupt handler for the Hibernation module interrupt.

Prototype:

```
void
HibernateIntUnregister(void)
```

Description:

Unregisters the interrupt handler in the system interrupt controller. The interrupt is disabled at the global level, and the interrupt handler will no longer be called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

11.2.2.12 HibernateIsActive

Checks to see if the Hibernation module is already powered up.

Prototype:

```
unsigned int
HibernateIsActive(void)
```

Description:

This function queries the control register to determine if the module is already active. This function can be called at a power-on reset to help determine if the reset is due to a wake from hibernation or a cold start. If the Hibernation module is already active, then it does not need to be re-enabled and its status can be queried immediately.

The software application should also use the HibernateIntStatus() function to read the raw interrupt status to determine the cause of the wake. The HibernateDataGet() function can be used to restore state. These combinations of functions can be used by the software to determine if the processor is waking from hibernation and the appropriate action to take as a result.

Returns:

Returns **true** if the module is already active, and **false** if not.

11.2.2.13 HibernateLowBatGet

Gets the currently configured low battery detection behavior.

Prototype:

```
unsigned long
HibernateLowBatGet(void)
```

Description:

Returns a value representing the currently configured low battery detection behavior. The return value will be one of the following:

- **HIBERNATE_LOW_BAT_DETECT** - detect a low battery condition.
- **HIBERNATE_LOW_BAT_ABORT** - detect a low battery condition, and abort hibernation if low battery is detected.

Returns:

Returns a value indicating the configured low battery detection.

11.2.2.14 HibernateLowBatSet

Configures the low battery detection.

Prototype:

```
void
HibernateLowBatSet(unsigned long ulLowBatFlags)
```

Parameters:

ulLowBatFlags specifies behavior of low battery detection.

Description:

Enables the low battery detection and whether hibernation is allowed if a low battery is detected. If low battery detection is enabled, then a low battery condition will be indicated in the raw interrupt status register, and can also trigger an interrupt. Optionally, hibernation can be aborted if a low battery is detected.

The *ulLowBatFlags* parameter is one of the following values:

- **HIBERNATE_LOW_BAT_DETECT** - detect a low battery condition.
- **HIBERNATE_LOW_BAT_ABORT** - detect a low battery condition, and abort hibernation if low battery is detected.

Returns:

None.

11.2.2.15 HibernateRequest

Requests hibernation mode.

Prototype:

```
void
HibernateRequest(void)
```

Description:

This function requests the Hibernation module to disable the external regulator, thus removing power from the processor and all peripherals. The Hibernation module will remain powered from the battery or auxiliary power supply.

The Hibernation module will re-enable the external regulator when one of the configured wake conditions occurs (such as RTC match or external **WAKE** pin). When the power is restored the processor will go through a normal power-on reset. The processor can retrieve saved state information with the HibernateDataGet() function. Prior to calling the function to request hibernation mode, the conditions for waking must have already been set by using the HibernateWakeSet() function.

Note that this function may return because some time may elapse before the power is actually removed, or it may not be removed at all. For this reason, the processor will continue to execute instructions for some time and the caller should be prepared for this function to return. There are various reasons why the power may not be removed. For example, if the HibernateLowBatSet() function was used to configure an abort if low battery is detected, then the power will not be removed if the battery voltage is too low. There may be other reasons, related to the external circuit design, that a request for hibernation may not actually occur.

For all these reasons, the caller must be prepared for this function to return. The simplest way to handle it is to just enter an infinite loop and wait for the power to be removed.

Returns:

None.

11.2.2.16 HibernateRTCDisable

Disables the RTC feature of the Hibernation module.

Prototype:

```
void
HibernateRTCDisable(void)
```

Description:

Disables the RTC in the Hibernation module. After calling this function the RTC features of the Hibernation module will not be available.

Returns:

None.

11.2.2.17 HibernateRTCEnable

Enables the RTC feature of the Hibernation module.

Prototype:

```
void
HibernateRTCEnable(void)
```

Description:

Enables the RTC in the Hibernation module. The RTC can be used to wake the processor from hibernation at a certain time, or to generate interrupts at certain times. This function must be called before using any of the RTC features of the Hibernation module.

Returns:

None.

11.2.2.18 HibernateRTCGet

Gets the value of the real time clock (RTC) counter.

Prototype:

```
unsigned long
HibernateRTCGet(void)
```

Description:

Gets the value of the RTC and returns it to the caller.

Returns:

Returns the value of the RTC.

11.2.2.19 HibernateRTCMatch0Get

Gets the value of the RTC match 0 register.

Prototype:

```
unsigned long
HibernateRTCMatch0Get(void)
```

Description:

Gets the value of the match 0 register for the RTC.

Returns:
Returns the value of the match register.

11.2.2.20 HibernateRTCMatch0Set

Sets the value of the RTC match 0 register.

Prototype:
```
void
HibernateRTCMatch0Set(unsigned long ulMatch)
```

Parameters:
ulMatch is the value for the match register.

Description:
Sets the match 0 register for the RTC. The Hibernation module can be configured to wake from hibernation, and/or generate an interrupt when the value of the RTC counter is the same as the match register.

Returns:
None.

11.2.2.21 HibernateRTCMatch1Get

Gets the value of the RTC match 1 register.

Prototype:
```
unsigned long
HibernateRTCMatch1Get(void)
```

Description:
Gets the value of the match 1 register for the RTC.

Returns:
Returns the value of the match register.

11.2.2.22 HibernateRTCMatch1Set

Sets the value of the RTC match 1 register.

Prototype:
```
void
HibernateRTCMatch1Set(unsigned long ulMatch)
```

Parameters:
ulMatch is the value for the match register.

Description:
Sets the match 1 register for the RTC. The Hibernation module can be configured to wake from hibernation, and/or generate an interrupt when the value of the RTC counter is the same as the match register.

Returns:
None.

11.2.2.23 HibernateRTCSet

Sets the value of the real time clock (RTC) counter.

Prototype:
```
void
HibernateRTCSet(unsigned long ulRTCValue)
```

Parameters:
ulRTCValue is the new value for the RTC.

Description:
Sets the value of the RTC. The RTC will count seconds if the hardware is configured correctly. The RTC must be enabled by calling HibernateRTCEnable() before calling this function.

Returns:
None.

11.2.2.24 HibernateRTCTrimGet

Gets the value of the RTC predivider trim register.

Prototype:
```
unsigned long
HibernateRTCTrimGet(void)
```

Description:
Gets the value of the pre-divider trim register. This function can be used to get the current value of the trim register prior to making an adjustment by using the HibernateRTCTrimSet() function.

Returns:
None.

11.2.2.25 HibernateRTCTrimSet

Sets the value of the RTC predivider trim register.

Prototype:
```
void
HibernateRTCTrimSet(unsigned long ulTrim)
```

Parameters:
ulTrim is the new value for the pre-divider trim register.

Description:
Sets the value of the pre-divider trim register. The input time source is divided by the pre-divider to achieve a one-second clock rate. Once every 64 seconds, the value of the pre-divider trim register is applied to the predivider to allow fine-tuning of the RTC rate, in order to make corrections to the rate. The software application can make adjustments to the predivider trim register to account for variations in the accuracy of the input time source. The nominal value is 0x7FFF, and it can be adjusted up or down in order to fine-tune the RTC rate.

Returns:
None.

11.2.2.26 HibernateWakeGet

Gets the currently configured wake conditions for the Hibernation module.

Prototype:
```
unsigned long
HibernateWakeGet(void)
```

Description:
Returns the flags representing the wake configuration for the Hibernation module. The return value will be a combination of the following flags:

- **HIBERNATE_WAKE_PIN** - wake when the external wake pin is asserted.
- **HIBERNATE_WAKE_RTC** - wake when one of the RTC matches occurs.

Returns:
Returns flags indicating the configured wake conditions.

11.2.2.27 HibernateWakeSet

Configures the wake conditions for the Hibernation module.

Prototype:
```
void
HibernateWakeSet(unsigned long ulWakeFlags)
```

Parameters:
ulWakeFlags specifies which conditions should be used for waking.

Description:
Enables the conditions under which the Hibernation module will wake. The *ulWakeFlags* parameter is the logical OR of any combination of the following:

- **HIBERNATE_WAKE_PIN** - wake when the external wake pin is asserted.
- **HIBERNATE_WAKE_RTC** - wake when one of the RTC matches occurs.

Returns:
None.

11.3 Programming Example

The following example shows how to determine if the processor reset is due to a wake from hibernation, and to restore saved state:

```
unsigned long ulStatus;
unsigned long ulNVData[64];

//
// Need to enable the hibernation peripheral after wake/reset, before using
// it.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);

//
// Determine if the Hibernation module is active.
//
if(HibernateIsActive())
{
    //
    // Read the status to determine cause of wake.
    //
    ulStatus = HibernateIntStatus(false);

    //
    // Test the status bits to see the cause.
    //
    if(ulStatus & HIBERNATE_INT_PIN_WAKE)
    {
        //
        // Wakeup was due to WAKE pin assertion.
        //
    }
    if(ulStatus & HIBERNATE_INT_RTC_MATCH_0)
    {
        //
        // Wakeup was due to RTC match0 register.
        //
    }

    //
    // Restore program state information that was saved prior to
    // hibernation.
    //
    HibernateDataGet(ulNVData, 64);

    //
    // Now that wakeup cause has been determined and state has been
    // restored, the program can proceed with normal processor and
    // peripheral initialization.
    //
}

//
// Hibernation module was not active so this is a cold power-up/reset.
//
else
{
    //
    // Perform normal power-on initialization.
    //
}
```

The following example shows how to set up the Hibernation module and initiate a hibernation with wake up at a future time:

```
unsigned long ulStatus;
unsigned long ulNVData[64];

//
// Need to enable the hibernation peripheral before using it.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);

//
// Enable clocking to the Hibernation module.
//
HibernateEnableExpClk(SysCtlClockGet());

//
// User-implemented delay here to allow crystal to power up and stabilize.
//

//
// Configure the clock source for Hibernation module, and enable the RTC
// feature.  This configuration is for a 4.194304 MHz crystal.
//
HibernateClockSelect(HIBERNATE_CLOCK_SEL_DIV128);
HibernateRTCEnable();

//
// Set the RTC to 0, or an initial value.  The RTC can be set once when the
// system is initialized after the cold-startup, and then left to run.  Or
// it can be initialized before every hibernate.
//
HibernateRTCSet(0);

//
// Set the match 0 register for 30 seconds from now.
//
HibernateRTCMatch0Set(HibernateRTCGet() + 30);

//
// Clear any pending status.
//
ulStatus = HibernateIntStatus(0);
HibernateIntClear(ulStatus);

//
// Save the program state information.  The state information will be
// stored in the ulNVData[] array.  It is not necessary to save the full 64
// words of data, only as much as is actually needed by the program.
//
HibernateDataSet(ulNVData, 64);

//
// Configure to wake on RTC match.
//
HibernateWakeSet(HIBERNATE_WAKE_RTC);

//
// Request hibernation.  The following call may return since it takes a
// finite amount of time for power to be removed.
//
HibernateRequest();

//
// Need a loop here to wait for the power to be removed.  Power will be
// removed while executing in this loop.
```

```
    //
    for(;;)
    {
    }
```

The following example shows how to use the Hibernation module RTC to generate an interrupt at a certain time:

```
//
// Handler for hibernate interrupts.
//
void
HibernateHandler(void)
{
    unsigned long ulStatus;

    //
    // Get the interrupt status, and clear any pending interrupts.
    //
    ulStatus = HibernateIntStatus(1);
    HibernateIntClear(ulStatus);

    //
    // Process the RTC match 0 interrupt.
    //
    if(ulStatus & HIBERNATE_INT_RTC_MATCH_0)
    {
        //
        // RTC match 0 interrupt actions go here.
        //
    }
}

//
// Main function.
//
int
main(void)
{
    //
    // System initialization code ...
    //

    //
    // Enable the Hibernation module.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);
    HibernateEnableExpClk(SysCtlClockGet());

    //
    // Wait an amount of time for the module to power up.
    //

    //
    // Configure the clock source for Hibernation module, and enable the
    // RTC feature.  This configuration is for the 4.194304 MHz crystal.
    //
    HibernateClockSelect(HIBERNATE_CLOCK_SEL_DIV128);
    HibernateRTCEnable();

    //
    // Set the RTC to an initial value.
    //
    HibernateRTCSet(0);
```

```
//
// Set Match 0 for 30 seconds from now.
//
HibernateRTCMatch0Set(HibernateRTCGet() + 30);

//
// Set up interrupts on the Hibernation module to enable the RTC match
// 0 interrupt.  Clear all pending interrupts and register the
// interrupt handler.
//
HibernateIntEnable(HIBERNATE_INT_RTC_MATCH_0);
HibernateIntClear(HIBERNATE_INT_PIN_WAKE | HIBERNATE_INT_LOW_BAT |
                  HIBERNATE_INT_RTC_MATCH_0 |
                  HIBERNATE_INT_RTC_MATCH_1);
HibernateIntRegister(HibernateHandler);

//
// Hibernate handler (above) will now be invoked in 30 seconds.
//

// ...
```

12 Inter-Integrated Circuit (I2C)

12.1 Introduction

The Inter-Integrated Circuit (I2C) API provides a set of functions for using the Stellaris I2C master and slave modules. Functions are provided to initialize the I2C modules, to send and receive data, obtain status, and to manage interrupts for the I2C modules.

The I2C master and slave modules provide the ability to communicate to other IC devices over an I2C bus. The I2C bus is specified to support devices that can both transmit and receive (write and read) data. Also, devices on the I2C bus can be designated as either a master or a slave. The Stellaris I2C modules support both sending and receiving data as either a master or a slave, and also support the simultaneous operation as both a master and a slave. Finally, the Stellaris I2C modules can operate at two speeds: Standard (100 kb/s) and Fast (400 kb/s).

Both the master and slave I2C modules can generate interrupts. The I2C master module will generate interrupts when a transmit or receive operation is completed (or aborted due to an error). The I2C slave module will generate interrupts when data has been sent or requested by a master.

12.1.1 Master Operations

When using this API to drive the I2C master module, the user must first initialize the I2C master module with a call to I2CMasterInitExpClk(). That function will set the bus speed and enable the master module.

The user may transmit or receive data after the successful initialization of the I2C master module. Data is transferred by first setting the slave address using I2CMasterSlaveAddrSet(). That function is also used to define whether the transfer is a send (a write to the slave from the master) or a receive (a read from the slave by the master). Then, if connected to an I2C bus that has multiple masters, the Stellaris I2C master must first call I2CMasterBusBusy() before attempting to initiate the desired transaction. After determining that the bus is not busy, if trying to send data, the user must call the I2CMasterDataPut() function. The transaction can then be initiated on the bus by calling the I2CMasterControl() function with any of the following commands:

- **I2C_MASTER_CMD_SINGLE_SEND**
- **I2C_MASTER_CMD_SINGLE_RECEIVE**
- **I2C_MASTER_CMD_BURST_SEND_START**
- **I2C_MASTER_CMD_BURST_RECEIVE_START**

Any of those commands will result in the master arbitrating for the bus, driving the start sequence onto the bus, and sending the slave address and direction bit across the bus. The remainder of the transaction can then be driven using either a polling or interrupt-driven method.

For the single send and receive cases, the polling method will involve looping on the return from I2CMasterBusy(). Once that function indicates that the I2C master is no longer busy, the bus transaction has been completed and can be checked for errors using I2CMasterErr(). If there are no

errors, then the data has been sent or is ready to be read using I2CMasterDataGet(). For the burst send and receive cases, the polling method also involves calling the I2CMasterControl() function for each byte transmitted or received (using either the **I2C_MASTER_CMD_BURST_SEND_CONT** or **I2C_MASTER_CMD_BURST_RECEIVE_CONT** commands), and for the last byte sent or received (using either the **I2C_MASTER_CMD_BURST_SEND_FINISH** or **I2C_MASTER_CMD_BURST_RECEIVE_FINISH** commands). If any error is detected during the burst transfer, the I2CMasterControl() function should be called using the appropriate stop command (**I2C_MASTER_CMD_BURST_SEND_ERROR_STOP** or **I2C_MASTER_CMD_BURST_RECEIVE_ERROR_STOP**).

For the interrupt-driven transaction, the user must register an interrupt handler for the I2C devices and enable the I2C master interrupt; the interrupt will occur when the master is no longer busy.

12.1.2 Slave Operations

When using this API to drive the I2C slave module, the user must first initialize the I2C slave module with a call to I2CSlaveInit(). This will enable the I2C slave module and initialize the slave's own address. After the initialization is complete, the user may poll the slave status using I2CSlaveStatus() to determine if a master requested a send or receive operation. Depending on the type of operation requested, the user can call I2CSlaveDataPut() or I2CSlaveDataGet() to complete the transaction. Alternatively, the I2C slave can handle transactions using an interrupt handler registered with I2CIntRegister(), and by enabling the I2C slave interrupt.

This driver is contained in `src/i2c.c`, with `src/i2c.h` containing the API definitions for use by applications.

12.2 API Functions

Functions

- void I2CIntRegister (unsigned long ulBase, void (*pfnHandler)(void))
- void I2CIntUnregister (unsigned long ulBase)
- tBoolean I2CMasterBusBusy (unsigned long ulBase)
- tBoolean I2CMasterBusy (unsigned long ulBase)
- void I2CMasterControl (unsigned long ulBase, unsigned long ulCmd)
- unsigned long I2CMasterDataGet (unsigned long ulBase)
- void I2CMasterDataPut (unsigned long ulBase, unsigned char ucData)
- void I2CMasterDisable (unsigned long ulBase)
- void I2CMasterEnable (unsigned long ulBase)
- unsigned long I2CMasterErr (unsigned long ulBase)
- void I2CMasterInitExpClk (unsigned long ulBase, unsigned long ulI2CClk, tBoolean bFast)
- void I2CMasterIntClear (unsigned long ulBase)
- void I2CMasterIntDisable (unsigned long ulBase)
- void I2CMasterIntEnable (unsigned long ulBase)
- tBoolean I2CMasterIntStatus (unsigned long ulBase, tBoolean bMasked)
- void I2CMasterSlaveAddrSet (unsigned long ulBase, unsigned char ucSlaveAddr, tBoolean bReceive)

- unsigned long I2CSlaveDataGet (unsigned long ulBase)
- void I2CSlaveDataPut (unsigned long ulBase, unsigned char ucData)
- void I2CSlaveDisable (unsigned long ulBase)
- void I2CSlaveEnable (unsigned long ulBase)
- void I2CSlaveInit (unsigned long ulBase, unsigned char ucSlaveAddr)
- void I2CSlaveIntClear (unsigned long ulBase)
- void I2CSlaveIntClearEx (unsigned long ulBase, unsigned long ulIntFlags)
- void I2CSlaveIntDisable (unsigned long ulBase)
- void I2CSlaveIntDisableEx (unsigned long ulBase, unsigned long ulIntFlags)
- void I2CSlaveIntEnable (unsigned long ulBase)
- void I2CSlaveIntEnableEx (unsigned long ulBase, unsigned long ulIntFlags)
- tBoolean I2CSlaveIntStatus (unsigned long ulBase, tBoolean bMasked)
- unsigned long I2CSlaveIntStatusEx (unsigned long ulBase, tBoolean bMasked)
- unsigned long I2CSlaveStatus (unsigned long ulBase)

12.2.1 Detailed Description

The I2C API is broken into three groups of functions: those that deal with interrupts, those that handle status and initialization, and those that deal with sending and receiving data.

The I2C master and slave interrupts are handled by the I2CIntRegister(), I2CIntUnregister(), I2CMasterIntEnable(), I2CMasterIntDisable(), I2CMasterIntClear(), I2CMasterIntStatus(), I2CSlaveIntEnable(), I2CSlaveIntDisable(), I2CSlaveIntClear(), I2CSlaveIntStatus(), I2CSlaveIntEnableEx(), I2CSlaveIntDisableEx(), I2CSlaveIntClearEx(), and I2CSlaveIntStatusEx() functions.

Status and initialization functions for the I2C modules are I2CMasterInitExpClk(), I2CMasterEnable(), I2CMasterDisable(), I2CMasterBusBusy(), I2CMasterBusy(), I2CMasterErr(), I2CSlaveInit(), I2CSlaveEnable(), I2CSlaveDisable(), and I2CSlaveStatus().

Sending and receiving data from the I2C modules are handled by the I2CMasterSlaveAddrSet(), I2CMasterControl(), I2CMasterDataGet(), I2CMasterDataPut(), I2CSlaveDataGet(), and I2CSlaveDataPut() functions.

The I2CMasterInit() API from previous versions of the peripheral driver library has been replaced by the I2CMasterInitExpClk() API. A macro has been provided in `i2c.h` to map the old API to the new API, allowing existing applications to link and run with the new API. It is recommended that new applications utilize the new API in favor of the old one.

12.2.2 Function Documentation

12.2.2.1 I2CIntRegister

Registers an interrupt handler for the I2C module.

Prototype:

```
void
I2CIntRegister(unsigned long ulBase,
               void (*pfnHandler)(void))
```

Parameters:

ulBase is the base address of the I2C Master module.

pfnHandler is a pointer to the function to be called when the I2C interrupt occurs.

Description:

This sets the handler to be called when an I2C interrupt occurs. This will enable the global interrupt in the interrupt controller; specific I2C interrupts must be enabled via I2CMasterIntEnable() and I2CSlaveIntEnable(). If necessary, it is the interrupt handler's responsibility to clear the interrupt source via I2CMasterIntClear() and I2CSlaveIntClear().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

12.2.2.2 I2CIntUnregister

Unregisters an interrupt handler for the I2C module.

Prototype:

```
void
I2CIntUnregister(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the I2C Master module.

Description:

This function will clear the handler to be called when an I2C interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

12.2.2.3 I2CMasterBusBusy

Indicates whether or not the I2C bus is busy.

Prototype:

```
tBoolean
I2CMasterBusBusy(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the I2C Master module.

Description:

This function returns an indication of whether or not the I2C bus is busy. This function can be used in a multi-master environment to determine if another master is currently using the bus.

Returns:
Returns **true** if the I2C bus is busy; otherwise, returns **false**.

12.2.2.4 I2CMasterBusy

Indicates whether or not the I2C Master is busy.

Prototype:
```
tBoolean
I2CMasterBusy(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the I2C Master module.

Description:
This function returns an indication of whether or not the I2C Master is busy transmitting or receiving data.

Returns:
Returns **true** if the I2C Master is busy; otherwise, returns **false**.

12.2.2.5 I2CMasterControl

Controls the state of the I2C Master module.

Prototype:
```
void
I2CMasterControl(unsigned long ulBase,
                 unsigned long ulCmd)
```

Parameters:
ulBase is the base address of the I2C Master module.
ulCmd command to be issued to the I2C Master module

Description:
This function is used to control the state of the Master module send and receive operations. The *ucCmd* parameter can be one of the following values:

- **I2C_MASTER_CMD_SINGLE_SEND**
- **I2C_MASTER_CMD_SINGLE_RECEIVE**
- **I2C_MASTER_CMD_BURST_SEND_START**
- **I2C_MASTER_CMD_BURST_SEND_CONT**
- **I2C_MASTER_CMD_BURST_SEND_FINISH**
- **I2C_MASTER_CMD_BURST_SEND_ERROR_STOP**
- **I2C_MASTER_CMD_BURST_RECEIVE_START**
- **I2C_MASTER_CMD_BURST_RECEIVE_CONT**
- **I2C_MASTER_CMD_BURST_RECEIVE_FINISH**
- **I2C_MASTER_CMD_BURST_RECEIVE_ERROR_STOP**

Returns:
None.

12.2.2.6 I2CMasterDataGet

Receives a byte that has been sent to the I2C Master.

Prototype:

```
unsigned long
I2CMasterDataGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the I2C Master module.

Description:

This function reads a byte of data from the I2C Master Data Register.

Returns:

Returns the byte received from by the I2C Master, cast as an unsigned long.

12.2.2.7 I2CMasterDataPut

Transmits a byte from the I2C Master.

Prototype:

```
void
I2CMasterDataPut(unsigned long ulBase,
                 unsigned char ucData)
```

Parameters:

ulBase is the base address of the I2C Master module.

ucData data to be transmitted from the I2C Master

Description:

This function will place the supplied data into I2C Master Data Register.

Returns:

None.

12.2.2.8 I2CMasterDisable

Disables the I2C master block.

Prototype:

```
void
I2CMasterDisable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the I2C Master module.

Description:

This will disable operation of the I2C master block.

Returns:

None.

12.2.2.9 I2CMasterEnable

Enables the I2C Master block.

Prototype:

```
void
I2CMasterEnable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the I2C Master module.

Description:

This will enable operation of the I2C Master block.

Returns:

None.

12.2.2.10 I2CMasterErr

Gets the error status of the I2C Master module.

Prototype:

```
unsigned long
I2CMasterErr(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the I2C Master module.

Description:

This function is used to obtain the error status of the Master module send and receive operations.

Returns:

Returns the error status, as one of **I2C_MASTER_ERR_NONE**, **I2C_MASTER_ERR_ADDR_ACK**, **I2C_MASTER_ERR_DATA_ACK**, or **I2C_MASTER_ERR_ARB_LOST**.

12.2.2.11 I2CMasterInitExpClk

Initializes the I2C Master block.

Prototype:

```
void
I2CMasterInitExpClk(unsigned long ulBase,
                    unsigned long ulI2CClk,
                    tBoolean bFast)
```

Parameters:

ulBase is the base address of the I2C Master module.

ulI2CClk is the rate of the clock supplied to the I2C module.

bFast set up for fast data transfers

Description:

This function initializes operation of the I2C Master block. Upon successful initialization of the I2C block, this function will have set the bus speed for the master, and will have enabled the I2C Master block.

If the parameter *bFast* is **true**, then the master block will be set up to transfer data at 400 kbps; otherwise, it will be set up to transfer data at 100 kbps.

The peripheral clock will be the same as the processor clock. This will be the value returned by SysCtlClockGet(), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to SysCtlClockGet()).

This function replaces the original I2CMasterInit() API and performs the same actions. A macro is provided in `i2c.h` to map the original API to this API.

Returns:

None.

12.2.2.12 I2CMasterIntClear

Clears I2C Master interrupt sources.

Prototype:

```
void
I2CMasterIntClear(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the I2C Master module.

Description:

The I2C Master interrupt source is cleared, so that it no longer asserts. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

12.2.2.13 I2CMasterIntDisable

Disables the I2C Master interrupt.

Prototype:

```
void
I2CMasterIntDisable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the I2C Master module.

Description:
Disables the I2C Master interrupt source.

Returns:
None.

12.2.2.14 I2CMasterIntEnable

Enables the I2C Master interrupt.

Prototype:
```
void
I2CMasterIntEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the I2C Master module.

Description:
Enables the I2C Master interrupt source.

Returns:
None.

12.2.2.15 I2CMasterIntStatus

Gets the current I2C Master interrupt status.

Prototype:
```
tBoolean
I2CMasterIntStatus(unsigned long ulBase,
                   tBoolean bMasked)
```

Parameters:
ulBase is the base address of the I2C Master module.
bMasked is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

Description:
This returns the interrupt status for the I2C Master module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:
The current interrupt status, returned as **true** if active or **false** if not active.

12.2.2.16 I2CMasterSlaveAddrSet

Sets the address that the I2C Master will place on the bus.

Prototype:

```
void
I2CMasterSlaveAddrSet(unsigned long ulBase,
                      unsigned char ucSlaveAddr,
                      tBoolean bReceive)
```

Parameters:

ulBase is the base address of the I2C Master module.

ucSlaveAddr 7-bit slave address

bReceive flag indicating the type of communication with the slave

Description:

This function will set the address that the I2C Master will place on the bus when initiating a transaction. When the *bReceive* parameter is set to **true**, the address will indicate that the I2C Master is initiating a read from the slave; otherwise the address will indicate that the I2C Master is initiating a write to the slave.

Returns:

None.

12.2.2.17 I2CSlaveDataGet

Receives a byte that has been sent to the I2C Slave.

Prototype:

```
unsigned long
I2CSlaveDataGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the I2C Slave module.

Description:

This function reads a byte of data from the I2C Slave Data Register.

Returns:

Returns the byte received from by the I2C Slave, cast as an unsigned long.

12.2.2.18 I2CSlaveDataPut

Transmits a byte from the I2C Slave.

Prototype:

```
void
I2CSlaveDataPut(unsigned long ulBase,
                unsigned char ucData)
```

Parameters:
ulBase is the base address of the I2C Slave module.
ucData data to be transmitted from the I2C Slave

Description:
This function will place the supplied data into I2C Slave Data Register.

Returns:
None.

12.2.2.19 I2CSlaveDisable

Disables the I2C slave block.

Prototype:
```
void
I2CSlaveDisable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the I2C Slave module.

Description:
This will disable operation of the I2C slave block.

Returns:
None.

12.2.2.20 I2CSlaveEnable

Enables the I2C Slave block.

Prototype:
```
void
I2CSlaveEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the I2C Slave module.

Description:
This will enable operation of the I2C Slave block.

Returns:
None.

12.2.2.21 I2CSlaveInit

Initializes the I2C Slave block.

Prototype:
```
void
I2CSlaveInit(unsigned long ulBase,
             unsigned char ucSlaveAddr)
```

Parameters:

ulBase is the base address of the I2C Slave module.

ucSlaveAddr 7-bit slave address

Description:

This function initializes operation of the I2C Slave block. Upon successful initialization of the I2C blocks, this function will have set the slave address and have enabled the I2C Slave block.

The parameter *ucSlaveAddr* is the value that will be compared against the slave address sent by an I2C master.

Returns:

None.

12.2.2.22 I2CSlaveIntClear

Clears I2C Slave interrupt sources.

Prototype:

```
void
I2CSlaveIntClear(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the I2C Slave module.

Description:

The I2C Slave interrupt source is cleared, so that it no longer asserts. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

12.2.2.23 I2CSlaveIntClearEx

Clears I2C Slave interrupt sources.

Prototype:

```
void
I2CSlaveIntClearEx(unsigned long ulBase,
                   unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the I2C Slave module.

ulIntFlags is a bit mask of the interrupt sources to be cleared.

Description:

The specified I2C Slave interrupt sources are cleared, so that they no longer assert. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to I2CSlaveIntEnableEx().

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

12.2.2.24 I2CSlaveIntDisable

Disables the I2C Slave interrupt.

Prototype:

```
void
I2CSlaveIntDisable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the I2C Slave module.

Description:

Disables the I2C Slave interrupt source.

Returns:

None.

12.2.2.25 I2CSlaveIntDisableEx

Disables individual I2C Slave interrupt sources.

Prototype:

```
void
I2CSlaveIntDisableEx(unsigned long ulBase,
                     unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the I2C Slave module.

ulIntFlags is the bit mask of the interrupt sources to be disabled.

Description:

Disables the indicated I2C Slave interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to I2CSlaveIntEnableEx().

Returns:
None.

12.2.2.26 I2CSlaveIntEnable

Enables the I2C Slave interrupt.

Prototype:
```
void
I2CSlaveIntEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the I2C Slave module.

Description:
Enables the I2C Slave interrupt source.

Returns:
None.

12.2.2.27 I2CSlaveIntEnableEx

Enables individual I2C Slave interrupt sources.

Prototype:
```
void
I2CSlaveIntEnableEx(unsigned long ulBase,
                    unsigned long ulIntFlags)
```

Parameters:
ulBase is the base address of the I2C Slave module.
ulIntFlags is the bit mask of the interrupt sources to be enabled.

Description:
Enables the indicated I2C Slave interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ulIntFlags* parameter is the logical OR of any of the following:

- **I2C_SLAVE_INT_STOP** - Stop condition detected interrupt
- **I2C_SLAVE_INT_START** - Start condition detected interrupt
- **I2C_SLAVE_INT_DATA** - Data interrupt

Returns:
None.

12.2.2.28 I2CSlaveIntStatus

Gets the current I2C Slave interrupt status.

Prototype:

```
tBoolean
I2CSlaveIntStatus(unsigned long ulBase,
                  tBoolean bMasked)
```

Parameters:

ulBase is the base address of the I2C Slave module.

bMasked is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

Description:

This returns the interrupt status for the I2C Slave module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

The current interrupt status, returned as **true** if active or **false** if not active.

12.2.2.29 I2CSlaveIntStatusEx

Gets the current I2C Slave interrupt status.

Prototype:

```
unsigned long
I2CSlaveIntStatusEx(unsigned long ulBase,
                    tBoolean bMasked)
```

Parameters:

ulBase is the base address of the I2C Slave module.

bMasked is false if the raw interrupt status is requested and true if the masked interrupt status is requested.

Description:

This returns the interrupt status for the I2C Slave module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status, enumerated as a bit field of values described in I2CSlaveIntEnableEx().

12.2.2.30 I2CSlaveStatus

Gets the I2C Slave module status

Prototype:

```
unsigned long
I2CSlaveStatus(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the I2C Slave module.

Description:

This function will return the action requested from a master, if any. Possible values are:

- **I2C_SLAVE_ACT_NONE**
- **I2C_SLAVE_ACT_RREQ**
- **I2C_SLAVE_ACT_TREQ**
- **I2C_SLAVE_ACT_RREQ_FBR**

Returns:

Returns **I2C_SLAVE_ACT_NONE** to indicate that no action has been requested of the I2C Slave module, **I2C_SLAVE_ACT_RREQ** to indicate that an I2C master has sent data to the I2C Slave module, **I2C_SLAVE_ACT_TREQ** to indicate that an I2C master has requested that the I2C Slave module send data, and **I2C_SLAVE_ACT_RREQ_FBR** to indicate that an I2C master has sent data to the I2C slave and the first byte following the slave's own address has been received.

12.3 Programming Example

The following example shows how to use the I2C API to send data as a master.

```
//
// Initialize Master and Slave
//
I2CMasterInitExpClk(I2C_MASTER_BASE, SysCtlClockGet(), true);

//
// Specify slave address
//
I2CMasterSlaveAddrSet(I2C_MASTER_BASE, 0x3B, false);

//
// Place the character to be sent in the data register
//
I2CMasterDataPut(I2C_MASTER_BASE, 'Q');

//
// Initiate send of character from Master to Slave
//
I2CMasterControl(I2C_MASTER_BASE, I2C_MASTER_CMD_SINGLE_SEND);

//
// Delay until transmission completes
//
while(I2CMasterBusBusy(I2C_MASTER_BASE))
{
}
```

13 Interrupt Controller (NVIC)

13.1 Introduction

The interrupt controller API provides a set of functions for dealing with the Nested Vectored Interrupt Controller (NVIC). Functions are provided to enable and disable interrupts, register interrupt handlers, and set the priority of interrupts.

The NVIC provides global interrupt masking, prioritization, and handler dispatching. This version of the Stellaris family supports thirty-two interrupt sources and eight priority levels. Individual interrupt sources can be masked, and the processor interrupt can be globally masked as well (without affecting the individual source masks).

The NVIC is tightly coupled with the Cortex-M3 microprocessor. When the processor responds to an interrupt, NVIC will supply the address of the function to handle the interrupt directly to the processor. This eliminates the need for a global interrupt handler that queries the interrupt controller to determine the cause of the interrupt and branch to the appropriate handler, reducing interrupt response time.

The interrupt prioritization in the NVIC allows higher priority interrupts to be handled before lower priority interrupts, as well as allowing preemption of lower priority interrupt handlers by higher priority interrupts. Again, this helps reduce interrupt response time (for example, a 1 ms system control interrupt is not held off by the execution of a lower priority 1 second housekeeping interrupt handler).

Sub-prioritization is also possible; instead of having N bits of preemptable prioritization, NVIC can be configured (via software) for N - M bits of preemptable prioritization and M bits of subpriority. In this scheme, two interrupts with the same preemptable prioritization but different subpriorities will not cause a preemption; tail chaining will instead be used to process the two interrupts back-to-back.

If two interrupts with the same priority (and subpriority if so configured) are asserted at the same time, the one with the lower interrupt number will be processed first. NVIC keeps track of the nesting of interrupt handlers, allowing the processor to return from interrupt context only once all nested and pending interrupts have been handled.

Interrupt handlers can be configured in one of two ways; statically at compile time or dynamically at run time. Static configuration of interrupt handlers is accomplished by editing the interrupt handler table in the application's startup code. When statically configured, the interrupts must be explicitly enabled in NVIC via IntEnable() before the processor will respond to the interrupt (in addition to any interrupt enabling required within the peripheral itself).

Alternatively, interrupts can be configured at run-time using IntRegister() (or the analog in each individual driver). When using IntRegister(), the interrupt must also be enabled as before; when using the analogue in each individual driver, IntEnable() is called by the driver and does not need to be call by the application.

Run-time configuration of interrupt handlers requires that the interrupt handler table be placed on a 1 kB boundary in SRAM (typically this would be at the beginning of SRAM). Failure to do so will result in an incorrect vector address being fetched in response to an interrupt. The vector table is

in a section called “vtable” and should be placed appropriately with a linker script. Tools that do not support linker scripts (such as the evaluation version of RV-MDK) therefore do not support run-time configuration of interrupt handlers (though the full version of RV-MDK does).

This driver is contained in `src/interrupt.c`, with `src/interrupt.h` containing the API definitions for use by applications.

13.2 API Functions

Functions

- void IntDisable (unsigned long ulInterrupt)
- void IntEnable (unsigned long ulInterrupt)
- tBoolean IntMasterDisable (void)
- tBoolean IntMasterEnable (void)
- long IntPriorityGet (unsigned long ulInterrupt)
- unsigned long IntPriorityGroupingGet (void)
- void IntPriorityGroupingSet (unsigned long ulBits)
- void IntPrioritySet (unsigned long ulInterrupt, unsigned char ucPriority)
- void IntRegister (unsigned long ulInterrupt, void (*pfnHandler)(void))
- void IntUnregister (unsigned long ulInterrupt)

13.2.1 Detailed Description

The primary function of the interrupt controller API is to manage the interrupt vector table used by the NVIC to dispatch interrupt requests. Registering an interrupt handler is a simple matter of inserting the handler address into the table. By default, the table is filled with pointers to an internal handler that loops forever; it is an error for an interrupt to occur when there is no interrupt handler registered to process it. Therefore, interrupt sources should not be enabled before a handler has been registered, and interrupt sources should be disabled before a handler is unregistered. Interrupt handlers are managed with IntRegister() and IntUnregister().

Each interrupt source can be individually enabled and disabled via IntEnable() and IntDisable(). The processor interrupt can be enabled and disabled via IntMasterEnable() and IntMasterDisable(); this does not affect the individual interrupt enable states. Masking of the processor interrupt can be utilized as a simple critical section (only NMI will interrupt the processor while the processor interrupt is disabled), though this will have adverse effects on the interrupt response time.

The priority of each interrupt source can be set and examined via IntPrioritySet() and IntPriorityGet(). The priority assignments are defined by the hardware; the upper N bits of the 8-bit priority are examined to determine the priority of an interrupt (for the Stellaris family, N is 3). This allows priorities to be defined without a real need to know the exact number of supported priorities; moving to a device with more or fewer priority bits will continue to treat the interrupt source with a similar level of priority. Smaller priority numbers correspond to higher interrupt priority, so 0 is the highest priority.

13.2.2 Function Documentation

13.2.2.1 IntDisable

Disables an interrupt.

Prototype:

```
void
IntDisable(unsigned long ulInterrupt)
```

Parameters:

ulInterrupt specifies the interrupt to be disabled.

Description:

The specified interrupt is disabled in the interrupt controller. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

Returns:

None.

13.2.2.2 IntEnable

Enables an interrupt.

Prototype:

```
void
IntEnable(unsigned long ulInterrupt)
```

Parameters:

ulInterrupt specifies the interrupt to be enabled.

Description:

The specified interrupt is enabled in the interrupt controller. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

Returns:

None.

13.2.2.3 IntMasterDisable

Disables the processor interrupt.

Prototype:

```
tBoolean
IntMasterDisable(void)
```

Description:

Prevents the processor from receiving interrupts. This does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

Note:
Previously, this function had no return value. As such, it was possible to include `interrupt.h` and call this function without having included `hw_types.h`. Now that the return is a `tBoolean`, a compiler error will occur in this case. The solution is to include `hw_types.h` before including `interrupt.h`.

Returns:
Returns **true** if interrupts were already disabled when the function was called or **false** if they were initially enabled.

13.2.2.4 IntMasterEnable

Enables the processor interrupt.

Prototype:
```
tBoolean
IntMasterEnable(void)
```

Description:
Allows the processor to respond to interrupts. This does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

Note:
Previously, this function had no return value. As such, it was possible to include `interrupt.h` and call this function without having included `hw_types.h`. Now that the return is a `tBoolean`, a compiler error will occur in this case. The solution is to include `hw_types.h` before including `interrupt.h`.

Returns:
Returns **true** if interrupts were disabled when the function was called or **false** if they were initially enabled.

13.2.2.5 IntPriorityGet

Gets the priority of an interrupt.

Prototype:
```
long
IntPriorityGet(unsigned long ulInterrupt)
```

Parameters:
ulInterrupt specifies the interrupt in question.

Description:
This function gets the priority of an interrupt. See IntPrioritySet() for a definition of the priority value.

Returns:
Returns the interrupt priority, or -1 if an invalid interrupt was specified.

13.2.2.6 IntPriorityGroupingGet

Gets the priority grouping of the interrupt controller.

Prototype:

```
unsigned long
IntPriorityGroupingGet(void)
```

Description:

This function returns the split between preemptable priority levels and subpriority levels in the interrupt priority specification.

Returns:

The number of bits of preemptable priority.

13.2.2.7 IntPriorityGroupingSet

Sets the priority grouping of the interrupt controller.

Prototype:

```
void
IntPriorityGroupingSet(unsigned long ulBits)
```

Parameters:

ulBits specifies the number of bits of preemptable priority.

Description:

This function specifies the split between preemptable priority levels and subpriority levels in the interrupt priority specification. The range of the grouping values are dependent upon the hardware implementation; on the Stellaris family, three bits are available for hardware interrupt prioritization and therefore priority grouping values of three through seven have the same effect.

Returns:

None.

13.2.2.8 IntPrioritySet

Sets the priority of an interrupt.

Prototype:

```
void
IntPrioritySet(unsigned long ulInterrupt,
               unsigned char ucPriority)
```

Parameters:

ulInterrupt specifies the interrupt in question.

ucPriority specifies the priority of the interrupt.

Description:

This function is used to set the priority of an interrupt. When multiple interrupts are asserted simultaneously, the ones with the highest priority are processed before the lower priority interrupts. Smaller numbers correspond to higher interrupt priorities; priority 0 is the highest interrupt priority.

The hardware priority mechanism will only look at the upper N bits of the priority level (where N is 3 for the Stellaris family), so any prioritization must be performed in those bits. The remaining bits can be used to sub-prioritize the interrupt sources, and may be used by the hardware priority mechanism on a future part. This arrangement allows priorities to migrate to different NVIC implementations without changing the gross prioritization of the interrupts.

Returns:

None.

13.2.2.9 IntRegister

Registers a function to be called when an interrupt occurs.

Prototype:

```
void
IntRegister(unsigned long ulInterrupt,
            void (*pfnHandler)(void))
```

Parameters:

ulInterrupt specifies the interrupt in question.

pfnHandler is a pointer to the function to be called.

Description:

This function is used to specify the handler function to be called when the given interrupt is asserted to the processor. When the interrupt occurs, if it is enabled (via IntEnable()), the handler function will be called in interrupt context. Since the handler function can preempt other code, care must be taken to protect memory or peripherals that are accessed by the handler and other non-handler code.

Note:

The use of this function (directly or indirectly via a peripheral driver interrupt register function) moves the interrupt vector table from flash to SRAM. Therefore, care must be taken when linking the application to ensure that the SRAM vector table is located at the beginning of SRAM; otherwise NVIC will not look in the correct portion of memory for the vector table (it requires the vector table be on a 1 kB memory alignment). Normally, the SRAM vector table is so placed via the use of linker scripts; some tool chains, such as the evaluation version of RV-MDK, do not support linker scripts and therefore will not produce a valid executable. See the discussion of compile-time versus run-time interrupt handler registration in the introduction to this chapter.

Returns:

None.

13.2.2.10 IntUnregister

Unregisters the function to be called when an interrupt occurs.

Prototype:

```
void
IntUnregister(unsigned long ulInterrupt)
```

Parameters:

ulInterrupt specifies the interrupt in question.

Description:

This function is used to indicate that no handler should be called when the given interrupt is asserted to the processor. The interrupt source will be automatically disabled (via IntDisable()) if necessary.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

13.3 Programming Example

The following example shows how to use the Interrupt Controller API to register an interrupt handler and enable the interrupt.

```
//
// The interrupt handler function.
//
extern void IntHandler(void);

//
// Register the interrupt handler function for interrupt 5.
//
IntRegister(5, IntHandler);

//
// Enable interrupt 5.
//
IntEnable(5);

//
// Enable interrupt 5.
//
IntMasterEnable();
```

14 Memory Protection Unit (MPU)

14.1 Introduction

The Memory Protection Unit (MPU) API provides functions to configure the MPU. The MPU is tightly coupled to the Cortex-M3 processor core and provides a means to establish access permissions on regions of memory.

Up to eight memory regions can be defined. Each region has a base address and a size. The size is specified as a power of 2 between 32 bytes and 4 GB, inclusive. The region's base address must be aligned to the size of the region. Each region also has access permissions. Code execution can be allowed or disallowed for a region. A region can be set for read-only access, read/write access, or no access for both privileged and user modes. This can be used to set up an environment where only kernel or system code can access certain hardware registers or sections of code.

The MPU creates 8 sub-regions within each region. Any sub-region or combination of sub-regions can be disabled, allowing creation of "holes" or complex overlaying regions with different permissions. The sub-regions can also be used to create an unaligned beginning or ending of a region by disabling one or more of the leading or trailing sub-regions.

Once the regions are defined and the MPU is enabled, any access violation of a region will cause a memory management fault, and the fault handler will be activated.

This driver is contained in `src/mpu.c`, with `src/mpu.h` containing the API definitions for use by applications.

14.2 API Functions

Functions

- void MPUDisable (void)
- void MPUEnable (unsigned long ulMPUConfig)
- void MPUIntRegister (void (*pfnHandler)(void))
- void MPUIntUnregister (void)
- unsigned long MPURegionCountGet (void)
- void MPURegionDisable (unsigned long ulRegion)
- void MPURegionEnable (unsigned long ulRegion)
- void MPURegionGet (unsigned long ulRegion, unsigned long *pulAddr, unsigned long *pulFlags)
- void MPURegionSet (unsigned long ulRegion, unsigned long ulAddr, unsigned long ulFlags)

14.2.1 Detailed Description

The MPU APIs provide a means to enable and configure the MPU and memory protection regions.

Generally, the memory protection regions should be defined before enabling the MPU. The regions can be configured by calling MPURegionSet() once for each region to be configured.

A region that is defined by MPURegionSet() can be initially enabled or disabled. If the region is not initially enabled, it can be enabled later by calling MPURegionEnable(). An enabled region can be disabled by calling MPURegionDisable(). When a region is disabled, its configuration is preserved as long as it is not overwritten. In this case it can be enabled again with MPURegionEnable() without the need to reconfigure the region.

Care must be taken when setting up a protection region using MPURegionSet(). The function will write to multiple registers and is not protected from interrupts. Therefore, it is possible that an interrupt which accesses a region may occur while that region is in the process of being changed. The safest way to protect against this is to make sure that a region is always disabled before making any changes. Otherwise, it is up to the caller to ensure that MPURegionSet() is always called from within code that cannot be interrupted, or from code that will not be affected if an interrupt occurs while the region attributes are being changed.

The attributes of a region that has already been programmed can be retrieved and saved using the MPURegionGet() function. This function is intended to save the attributes in a format that can be used later to reload the region using the MPURegionSet() function. Note that the enable state of the region is saved with the attributes and will take effect when the region is reloaded.

When one or more regions are defined, the MPU can be enabled by calling MPUEnable(). This turns on the MPU and also defines the behavior in privileged mode and in the Hard Fault and NMI fault handlers. The MPU can be configured so that when in privileged mode and no regions are enabled, a default memory map is applied. If this feature is not enabled, then a memory management fault is generated if the MPU is enabled and no regions are configured and enabled. The MPU can also be set to use a default memory map when in the Hard Fault or NMI handlers, instead of using the configured regions. All of these features are selected when calling MPUEnable(). When the MPU is enabled, it can be disabled by calling MPUDisable().

Finally, if the application is using run-time interrupt registration (see IntRegister()), then the function MPUIntRegister() can be used to install the fault handler which will be called whenever a memory protection violation occurs. This function will also enable the fault handler. If compile-time interrupt registration is used, then the IntEnable() function with the parameter **FAULT_MPU** must be used to enable the memory management fault handler. When the memory management fault handler has been installed with MPUIntRegister(), it can be removed by calling MPUIntUnregister().

14.2.2 Function Documentation

14.2.2.1 MPUDisable

Disables the MPU for use.

Prototype:

```
void
MPUDisable(void)
```

Description:

This function disables the Cortex-M3 memory protection unit. When the MPU is disabled, the

default memory map is used and memory management faults are not generated.

Returns:
None.

14.2.2.2 MPUEnable

Enables and configures the MPU for use.

Prototype:
```
void
MPUEnable(unsigned long ulMPUConfig)
```

Parameters:
ulMPUConfig is the logical OR of the possible configurations.

Description:
This function enables the Cortex-M3 memory protection unit. It also configures the default behavior when in privileged mode and while handling a hard fault or NMI. Prior to enabling the MPU, at least one region must be set by calling MPURegionSet() or else by enabling the default region for privileged mode by passing the **MPU_CONFIG_PRIV_DEFAULT** flag to MPUEnable(). Once the MPU is enabled, a memory management fault will be generated for any memory access violations.

The *ulMPUConfig* parameter should be the logical OR of any of the following:

- **MPU_CONFIG_PRIV_DEFAULT** enables the default memory map when in privileged mode and when no other regions are defined. If this option is not enabled, then there must be at least one valid region already defined when the MPU is enabled.
- **MPU_CONFIG_HARDFLT_NMI** enables the MPU while in a hard fault or NMI exception handler. If this option is not enabled, then the MPU is disabled while in one of these exception handlers and the default memory map is applied.
- **MPU_CONFIG_NONE** chooses none of the above options. In this case, no default memory map is provided in privileged mode, and the MPU will not be enabled in the fault handlers.

Returns:
None.

14.2.2.3 MPUIntRegister

Registers an interrupt handler for the memory management fault.

Prototype:
```
void
MPUIntRegister(void (*pfnHandler)(void))
```

Parameters:
pfnHandler is a pointer to the function to be called when the memory management fault occurs.

Description:
This sets and enables the handler to be called when the MPU generates a memory management fault due to a protection region access violation.

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

14.2.2.4 MPUIntUnregister

Unregisters an interrupt handler for the memory management fault.

Prototype:
```
void
MPUIntUnregister(void)
```

Description:
This function will disable and clear the handler to be called when a memory management fault occurs.

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

14.2.2.5 MPURegionCountGet

Gets the count of regions supported by the MPU.

Prototype:
```
unsigned long
MPURegionCountGet(void)
```

Description:
This function is used to get the number of regions that are supported by the MPU. This is the total number that are supported, including regions that are already programmed.

Returns:
The number of memory protection regions that are available for programming using MPURegionSet().

14.2.2.6 MPURegionDisable

Disables a specific region.

Prototype:

```
void
MPURegionDisable(unsigned long ulRegion)
```

Parameters:

ulRegion is the region number to disable.

Description:

This function is used to disable a previously enabled memory protection region. The region will remain configured if it is not overwritten with another call to MPURegionSet(), and can be enabled again by calling MPURegionEnable().

Returns:

None.

14.2.2.7 MPURegionEnable

Enables a specific region.

Prototype:

```
void
MPURegionEnable(unsigned long ulRegion)
```

Parameters:

ulRegion is the region number to enable.

Description:

This function is used to enable a memory protection region. The region should already be set up with the MPURegionSet() function. Once enabled, the memory protection rules of the region will be applied and access violations will cause a memory management fault.

Returns:

None.

14.2.2.8 MPURegionGet

Gets the current settings for a specific region.

Prototype:

```
void
MPURegionGet(unsigned long ulRegion,
             unsigned long *pulAddr,
             unsigned long *pulFlags)
```

Parameters:

ulRegion is the region number to get.

pulAddr points to storage for the base address of the region.

pulFlags points to the attribute flags for the region.

Description:

This function retrieves the configuration of a specific region. The meanings and format of the parameters is the same as that of the MPURegionSet() function.

This function can be used to save the configuration of a region for later use with the MPURegionSet() function. The region's enable state will be preserved in the attributes that are saved.

Returns:

None.

14.2.2.9 MPURegionSet

Sets up the access rules for a specific region.

Prototype:

```
void
MPURegionSet(unsigned long ulRegion,
             unsigned long ulAddr,
             unsigned long ulFlags)
```

Parameters:

ulRegion is the region number to set up.

ulAddr is the base address of the region. It must be aligned according to the size of the region specified in ulFlags.

ulFlags is a set of flags to define the attributes of the region.

Description:

This function sets up the protection rules for a region. The region has a base address and a set of attributes including the size, which must be a power of 2. The base address parameter, *ulAddr*, must be aligned according to the size.

The *ulFlags* parameter is the logical OR of all of the attributes of the region. It is a combination of choices for region size, execute permission, read/write permissions, disabled sub-regions, and a flag to determine if the region is enabled.

The size flag determines the size of a region, and must be one of the following:

- **MPU_RGN_SIZE_32B**
- **MPU_RGN_SIZE_64B**
- **MPU_RGN_SIZE_128B**
- **MPU_RGN_SIZE_256B**
- **MPU_RGN_SIZE_512B**
- **MPU_RGN_SIZE_1K**
- **MPU_RGN_SIZE_2K**
- **MPU_RGN_SIZE_4K**
- **MPU_RGN_SIZE_8K**
- **MPU_RGN_SIZE_16K**
- **MPU_RGN_SIZE_32K**
- **MPU_RGN_SIZE_64K**
- **MPU_RGN_SIZE_128K**
- **MPU_RGN_SIZE_256K**

- **MPU_RGN_SIZE_512K**
- **MPU_RGN_SIZE_1M**
- **MPU_RGN_SIZE_2M**
- **MPU_RGN_SIZE_4M**
- **MPU_RGN_SIZE_8M**
- **MPU_RGN_SIZE_16M**
- **MPU_RGN_SIZE_32M**
- **MPU_RGN_SIZE_64M**
- **MPU_RGN_SIZE_128M**
- **MPU_RGN_SIZE_256M**
- **MPU_RGN_SIZE_512M**
- **MPU_RGN_SIZE_1G**
- **MPU_RGN_SIZE_2G**
- **MPU_RGN_SIZE_4G**

The execute permission flag must be one of the following:

- **MPU_RGN_PERM_EXEC** enables the region for execution of code
- **MPU_RGN_PERM_NOEXEC** disables the region for execution of code

The read/write access permissions are applied separately for the privileged and user modes. The read/write access flags must be one of the following:

- **MPU_RGN_PERM_PRV_NO_USR_NO** - no access in privileged or user mode
- **MPU_RGN_PERM_PRV_RW_USR_NO** - privileged read/write, user no access
- **MPU_RGN_PERM_PRV_RW_USR_RO** - privileged read/write, user read-only
- **MPU_RGN_PERM_PRV_RW_USR_RW** - privileged read/write, user read/write
- **MPU_RGN_PERM_PRV_RO_USR_NO** - privileged read-only, user no access
- **MPU_RGN_PERM_PRV_RO_USR_RO** - privileged read-only, user read-only

The region is automatically divided into 8 equally-sized sub-regions by the MPU. Sub-regions can only be used in regions of size 256 bytes or larger. Any of these 8 sub-regions can be disabled. This allows for creation of "holes" in a region which can be left open, or overlaid by another region with different attributes. Any of the 8 sub-regions can be disabled with a logical OR of any of the following flags:

- **MPU_SUB_RGN_DISABLE_0**
- **MPU_SUB_RGN_DISABLE_1**
- **MPU_SUB_RGN_DISABLE_2**
- **MPU_SUB_RGN_DISABLE_3**
- **MPU_SUB_RGN_DISABLE_4**
- **MPU_SUB_RGN_DISABLE_5**
- **MPU_SUB_RGN_DISABLE_6**
- **MPU_SUB_RGN_DISABLE_7**

Finally, the region can be initially enabled or disabled with one of the following flags:

- **MPU_RGN_ENABLE**
- **MPU_RGN_DISABLE**

As an example, to set a region with the following attributes: size of 32 KB, execution enabled, read-only for both privileged and user, one sub-region disabled, and initially enabled; the *ulFlags* parameter would have the following value:

```
(MPU_RG_SIZE_32K | MPU_RGN_PERM_EXEC | MPU_RGN_PERM_PRV_RO_USR_RO |
MPU_SUB_RGN_DISABLE_2 | MPU_RGN_ENABLE)
```

Note:
This function will write to multiple registers and is not protected from interrupts. It is possible that an interrupt which accesses a region may occur while that region is in the process of being changed. The safest way to handle this is to disable a region before changing it. Refer to the discussion of this in the API Detailed Description section.

Returns:
None.

14.3 Programming Example

The following example sets up a basic set of protection regions to provide the following:

- a 28 KB region in flash for read-only code execution
- 32 KB of RAM for read-write access in privileged and user modes
- an additional 8 KB of RAM for use only in privileged mode
- 1 MB of peripheral space for access only in privileged mode, except for a 128 KB hole that is not accessible at all, and another 128 KB region within that is accessible from user mode

```
//
// Define a 28 KB region of flash from 0x00000000 to 0x00007000.  The
// region will be executable, and read-only for both privileged and user
// modes.  To set up the region, a 32 KB region (#0) will be defined
// starting at address 0, and then a 4 KB hole removed at the end by
// disabling the last sub-region.  The region will be initially enabled.
//
MPURegionSet(0, 0,
             MPU_RGN_SIZE_32K |
             MPU_RGN_PERM_EXEC |
             MPU_RGN_PERM_PRV_RO_USR_RO |
             MPU_SUB_RGN_DISABLE_7 |
             MPU_RGN_ENABLE);

//
// Define a 32 KB region (#1) of RAM from 0x20000000 to 0x20008000.  The
// region will not be executable, and will be read/write access for
// privileged and user modes.
//
MPURegionSet(1, 0x20000000,
             MPU_RGN_SIZE_32K |
             MPU_RGN_PERM_NOEXEC |
             MPU_RGN_PERM_PRV_RW_USR_RW |
             MPU_RGN_ENABLE);

//
// Define an additional 8 KB region (#2) in RAM from 0x20008000 to
// 0x2000A000, which will be read/write accessible only from privileged
// mode.  This region will be initially disabled, to be enabled later.
//
```

```
MPURegionSet(2, 0x20008000,
             MPU_RGN_SIZE_8K |
             MPU_RGN_PERM_NOEXEC |
             MPU_RGN_PERM_PRV_RW_USR_NO|
             MPU_RGN_DISABLE);

//
// Define a region (#3) in peripheral space from 0x40000000 to 0x40100000
// (1 MB).  This region is accessible only in privileged mode.  There is a
// an area from 0x40020000 to 0x40040000 that has no peripherals and is not
// accessible at all.  This is created by disabling the second sub-region
// (1) and creating a hole.  Further, there is an area from 0x40080000 to
// 0x400A0000 that should be accessible from user mode as well.  This is
// created by disabling the fifth sub-region (4), and overlaying an
// additional region (#4) in that space with the appropriate permissions.
//
MPURegionSet(3, 0x40000000,
             MPU_RGN_SIZE_1M |
             MPU_RGN_PERM_NOEXEC |
             MPU_RGN_PERM_PRV_RW_USR_NO |
             MPU_SUB_RGN_DISABLE_1 | MPU_SUB_RGN_DISABLE_4 |
             MPU_RGN_ENABLE);
MPURegionSet(4, 0x40080000,
             MPU_RGN_SIZE_128K |
             MPU_RGN_PERM_NOEXEC |
             MPU_RGN_PERM_PRV_RW_USR_RW |
             MPU_RGN_ENABLE);

//
// In this example, compile-time registration of interrupts is used, so the
// handler does not need to be registered.  However, it does need to be
// enabled.
//
IntEnable(FAULT_MPU);

//
// When setting up the regions, region 2 was initially disabled for some
// reason.  At some point it needs to be enabled.
//
MPURegionEnable(2);

//
// Now the MPU will be enabled.  It will be configured so that a default
// map is available in privileged mode if no regions are defined.  The MPU
// will not be enabled for the hard fault and NMI handlers, which means a
// default map will be used whenever these handlers are active, effectively
// giving the fault handlers access to all of memory without any
// protection.
//
MPUEnable(MPU_CONFIG_PRIV_DEFAULT);

//
// At this point the MPU is configured and enabled and if any code causes
// an access violation, the memory management fault will occur.
//
```

The following example shows how to save and restore region configurations.

```
//
// The following arrays provide space for saving the address and
// attributes for 4 region configurations.
//
unsigned long ulRegionAddr[4];
unsigned long ulRegionAttr[4];
```

```
...

//
// At some point in the system code, we want to save the state of 4 regions
// (0-3).
//
for(uIdx = 0; uIdx < 4; uIdx++)
{
    MPURegionGet(uIdx, &ulRegionAddr[uIdx], &ulRegionAttr[uIdx]);
}

...

//
// At some other point, the previously saved regions should be restored.
//
for(uIdx = 0; uIdx < 4; uIdx++)
{
    MPURegionSet(uIdx, ulRegionAddr[uIdx], ulRegionAttr[uIdx]);
}
```

15 Peripheral Pin Mapping

15.1 Introduction

The peripheral pin mapping functions provide an easy method of configuring a peripheral pin without having to know which GPIO pin is shared with the peripheral pin. This makes peripheral pin configuration easier (and clearer) since the pin can be specified by the peripheral pin name instead of the GPIO name (which may be error prone).

The mapping of peripheral pins to GPIO pins varies from part to part, meaning that the associated definitions change based on the part being used. The part to be used can be specified in two ways; either via an explicit `#define` in the source code or via a definition provided to the compiler. Using a `#define` is very direct, but not very flexible. Using a definition provided to the compiler is not as explicit (since it does not appear clearly in the source code) but is much more flexible. The real value of the peripheral pin mapping functions is the ability to share a piece of peripheral configuration/control code between projects that utilize different parts; if the part definition is provided to the compiler instead of in the source code, each project can provide its own definition and the code will automatically reconfigure itself based on the target part.

Since the peripheral pin mapping functions configure a single pin at a time, it may be more efficient to use the GPIOPinType∗() functions instead of the PinType∗() functions, although this requires explicit knowledge of the GPIO pin(s) to be used. For example, it will take four PinTypeSSI() calls to configure the four pins on the SSI peripheral, but this could be done with a single call to GPIOPinTypeSSI() if the pins are all in the same GPIO module. But using GPIOPinType∗() instead of PinType∗() results in the code no longer automatically reconfiguring itself (without the use of explicit conditionals in the code, of course).

This driver is contained in `src/pin_map.h`.

15.2 API Functions

Functions

- void PeripheralEnable (unsigned long ulName)
- void PinTypeADC (unsigned long ulName)
- void PinTypeCAN (unsigned long ulName)
- void PinTypeComparator (unsigned long ulName)
- void PinTypeI2C (unsigned long ulName)
- void PinTypePWM (unsigned long ulName)
- void PinTypeQEI (unsigned long ulName)
- void PinTypeSSI (unsigned long ulName)
- void PinTypeTimer (unsigned long ulName)
- void PinTypeUART (unsigned long ulName)

- void PinTypeUSBDigital (unsigned long ulName)

15.2.1 Detailed Description

The peripheral pin mapping functions require that the part being used be specified by a define of the `PART_LM3Sxxx` form. The `xxx` portion is replaced with the part number of the part being used; for example, if using the LM3S6965 microcontroller, the define will be `PART_LM3S6965`. This must be defined before `pin_map.h` is included by the source code.

15.2.2 Function Documentation

15.2.2.1 PeripheralEnable

Enables the peripheral port used by the given pin.

Prototype:

```
void
PeripheralEnable(unsigned long ulName)
```

Parameters:

ulName is one of the valid names for a pin.

Description:

This function takes one of the valid names for a pin function and enables the peripheral port for that pin depending on the part that is defined.

Any valid pin name can be used.

See also:

SysCtlPeripheralEnable() in order to enable a single port when multiple pins are on the same port.

Returns:

None.

15.2.2.2 PinTypeADC

Configures the specified ADC pin to function as an ADC pin.

Prototype:

```
void
PinTypeADC(unsigned long ulName)
```

Parameters:

ulName is one of the valid names for the ADC pins.

Description:

This function takes on of the valid names for an ADC pin and configures the pin for its ADC functionality depending on the part that is defined.

The valid names for the pins are as follows: **ADC0**, **ADC1**, **ADC2**, **ADC3**, **ADC4**, **ADC5**, **ADC6**, or **ADC7**.

See also:
GPIOPinTypeADC() in order to configure multiple ADC pins at once.

Returns:
None.

15.2.2.3 PinTypeCAN

Configures the specified CAN pin to function as a CAN pin.

Prototype:
```
void
PinTypeCAN(unsigned long ulName)
```

Parameters:
ulName is one of the valid names for the CAN pins.

Description:
This function takes one of the valid names for a CAN pin and configures the pin for its CAN functionality depending on the part that is defined.

The valid names for the pins are as follows: **CAN0RX**, **CAN0TX**, **CAN1RX**, **CAN1TX**, **CAN2RX**, or **CAN2TX**.

See also:
GPIOPinTypeCAN() in order to configure multiple CAN pins at once.

Returns:
None.

15.2.2.4 PinTypeComparator

Configures the specified comparator pin to function as a comparator pin.

Prototype:
```
void
PinTypeComparator(unsigned long ulName)
```

Parameters:
ulName is one of the valid names for the Comparator pins.

Description:
This function takes one of the valid names for a comparator pin and configures the pin for its comparator functionality depending on the part that is defined.

The valid names for the pins are as follows: **C0_MINUS**, **C0_PLUS**, **C1_MINUS**, **C1_PLUS**, **C2_MINUS**, or **C2_PLUS**.

See also:
GPIOPinTypeComparator() in order to configure multiple comparator pins at once.

Returns:
None.

15.2.2.5 PinTypeI2C

Configures the specified I2C pin to function as an I2C pin.

Prototype:
```
void
PinTypeI2C(unsigned long ulName)
```

Parameters:
ulName is one of the valid names for the I2C pins.

Description:
This function takes one of the valid names for an I2C pin and configures the pin for its I2C functionality depending on the part that is defined.

The valid names for the pins are as follows: **I2C0SCL**, **I2C0SDA**, **I2C1SCL**, or **I2C1SDA**.

See also:
GPIOPinTypeI2C() in order to configure multiple I2C pins at once.

Returns:
None.

15.2.2.6 PinTypePWM

Configures the specified PWM pin to function as a PWM pin.

Prototype:
```
void
PinTypePWM(unsigned long ulName)
```

Parameters:
ulName is one of the valid names for the PWM pins.

Description:
This function takes one of the valid names for a PWM pin and configures the pin for its PWM functionality depending on the part that is defined.

The valid names for the pins are as follows: **PWM0**, **PWM1**, **PWM2**, **PWM3**, **PWM4**, **PWM5**, or **FAULT**.

See also:
GPIOPinTypePWM() in order to configure multiple PWM pins at once.

Returns:
None.

15.2.2.7 PinTypeQEI

Configures the specified QEI pin to function as a QEI pin.

Prototype:

```
void
PinTypeQEI(unsigned long ulName)
```

Parameters:

ulName is one of the valid names for the QEI pins.

Description:

This function takes one of the valid names for a QEI pin and configures the pin for its QEI functionality depending on the part that is defined.

The valid names for the pins are as follows: **PHA0**, **PHB0**, **IDX0**, **PHA1**, **PHB1**, or **IDX1**.

See also:

GPIOPinTypeQEI() in order to configure multiple QEI pins at once.

Returns:

None.

15.2.2.8 PinTypeSSI

Configures the specified SSI pin to function as an SSI pin.

Prototype:

```
void
PinTypeSSI(unsigned long ulName)
```

Parameters:

ulName is one of the valid names for the SSI pins.

Description:

This function takes one of the valid names for an SSI pin and configures the pin for its SSI functionality depending on the part that is defined.

The valid names for the pins are as follows: **SSI0CLK**, **SSI0FSS**, **SSI0RX**, **SSI0TX**, **SSI1CLK**, **SSI1FSS**, **SSI1RX**, or **SSI1TX**.

See also:

GPIOPinTypeSSI() in order to configure multiple SSI pins at once.

Returns:

None.

15.2.2.9 PinTypeTimer

Configures the specified Timer pin to function as a Timer pin.

Prototype:

```
void
PinTypeTimer(unsigned long ulName)
```

Parameters:

ulName is one of the valid names for the Timer pins.

Description:

This function takes one of the valid names for a Timer pin and configures the pin for its Timer functionality depending on the part that is defined.

The valid names for the pins are as follows: **CCP0**, **CCP1**, **CCP2**, **CCP3**, **CCP4**, **CCP5**, **CCP6**, or **CCP7**.

See also:

GPIOPinTypeTimer() in order to configure multiple CCP pins at once.

Returns:

None.

15.2.2.10 PinTypeUART

Configures the specified UART pin to function as a UART pin.

Prototype:

```
void
PinTypeUART(unsigned long ulName)
```

Parameters:

ulName is one of the valid names for the UART pins.

Description:

This function takes one of the valid names for a UART pin and configures the pin for its UART functionality depending on the part that is defined.

The valid names for the pins are as follows: **U0RX**, **U0TX**, **U1RX**, **U1TX**, **U2RX**, or **U2TX**.

See also:

GPIOPinTypeUART() in order to configure multiple UART pins at once.

Returns:

None.

15.2.2.11 PinTypeUSBDigital

Configures the specified USB digital pin to function as a USB pin.

Prototype:

```
void
PinTypeUSBDigital(unsigned long ulName)
```

Parameters:

ulName is one of the valid names for a USB digital pin.

Description:

This function takes one of the valid names for a USB digital pin and configures the pin for its USB functionality depending on the part that is defined.

The valid names for the pins are as follows: **EPEN** or **PFAULT**.

See also:

GPIOPinTypeUSBDigital() in order to configure multiple USB pins at once.

Returns:

None.

15.3 Programming Example

This example shows the difference in code when configuring a PWM pin on two different parts in the same application. In this case, the PWM0 pin is actually on a different GPIO port on the two parts and requires special conditional code if the GPIOPinTypePWM() function is used directly. Instead, if PinTypePWM() is used, then the code can remain the same and only the part definition in the project file needs to change.

Example for PWM0 pin configuration using PinTypePWM():

```
    ...

    //
    // Configure the pin for use as a PWM pin.
    //
    PinTypePWM(PWM0);

    ...
```

Example for PWM0 pin configuration using GPIOPinTypePWM():

```
    ...

#ifdef LM3S2110
    //
    // Configure the pin for use as a PWM pin.
    //
    GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_0);
#endif
#ifdef LM3S2620
    //
    // Configure the pin for use as a PWM pin.
    //
    GPIOPinTypeTimer(GPIO_PORTG_BASE, GPIO_PIN_0);
#endif

    ...
```

16 Pulse Width Modulator (PWM)

16.1 Introduction

Each instance of a Stellaris PWM module provides three instances of a PWM generator block, and an output control block. Each generator block has two PWM output signals, which can be operated independently, or as a pair of signals with dead band delays inserted. Each generator block also has an interrupt output and a trigger output. The control block determines the polarity of the PWM signals, and which signals are passed through to the pins.

Some of the features of the Stellaris PWM module are:

- Three generator blocks, each containing
  - One 16-bit down or up/down counter
  - Two comparators
  - PWM generator
  - Dead band generator
- Control block
  - PWM output enable
  - Output polarity control
  - Synchronization
  - Fault handling
  - Interrupt status

This driver is contained in `src/pwm.c`, with `src/pwm.h` containing the API definitions for use by applications.

16.2 API Functions

Functions

- void PWMDeadBandDisable (unsigned long ulBase, unsigned long ulGen)
- void PWMDeadBandEnable (unsigned long ulBase, unsigned long ulGen, unsigned short usRise, unsigned short usFall)
- void PWMFaultIntClear (unsigned long ulBase)
- void PWMFaultIntClearExt (unsigned long ulBase, unsigned long ulFaultInts)
- void PWMFaultIntRegister (unsigned long ulBase, void (*pfnIntHandler)(void))
- void PWMFaultIntUnregister (unsigned long ulBase)
- void PWMGenConfigure (unsigned long ulBase, unsigned long ulGen, unsigned long ulConfig)
- void PWMGenDisable (unsigned long ulBase, unsigned long ulGen)

- void PWMGenEnable (unsigned long ulBase, unsigned long ulGen)
- void PWMGenFaultClear (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup, unsigned long ulFaultTriggers)
- void PWMGenFaultConfigure (unsigned long ulBase, unsigned long ulGen, unsigned long ulMinFaultPeriod, unsigned long ulFaultSenses)
- unsigned long PWMGenFaultStatus (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup)
- unsigned long PWMGenFaultTriggerGet (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup)
- void PWMGenFaultTriggerSet (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup, unsigned long ulFaultTriggers)
- void PWMGenIntClear (unsigned long ulBase, unsigned long ulGen, unsigned long ulInts)
- void PWMGenIntRegister (unsigned long ulBase, unsigned long ulGen, void (*pfnIntHandler)(void))
- unsigned long PWMGenIntStatus (unsigned long ulBase, unsigned long ulGen, tBoolean bMasked)
- void PWMGenIntTrigDisable (unsigned long ulBase, unsigned long ulGen, unsigned long ulIntTrig)
- void PWMGenIntTrigEnable (unsigned long ulBase, unsigned long ulGen, unsigned long ulIntTrig)
- void PWMGenIntUnregister (unsigned long ulBase, unsigned long ulGen)
- unsigned long PWMGenPeriodGet (unsigned long ulBase, unsigned long ulGen)
- void PWMGenPeriodSet (unsigned long ulBase, unsigned long ulGen, unsigned long ulPeriod)
- void PWMIntDisable (unsigned long ulBase, unsigned long ulGenFault)
- void PWMIntEnable (unsigned long ulBase, unsigned long ulGenFault)
- unsigned long PWMIntStatus (unsigned long ulBase, tBoolean bMasked)
- void PWMOutputFault (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bFaultSuppress)
- void PWMOutputFaultLevel (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bDriveHigh)
- void PWMOutputInvert (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bInvert)
- void PWMOutputState (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bEnable)
- unsigned long PWMPulseWidthGet (unsigned long ulBase, unsigned long ulPWMOut)
- void PWMPulseWidthSet (unsigned long ulBase, unsigned long ulPWMOut, unsigned long ulWidth)
- void PWMSyncTimeBase (unsigned long ulBase, unsigned long ulGenBits)
- void PWMSyncUpdate (unsigned long ulBase, unsigned long ulGenBits)

16.2.1 Detailed Description

These are a group of functions for performing high-level operations on PWM modules. Although Stellaris only has one PWM module, these functions are defined to support using multiple instances of PWM modules.

The following functions provide the user with a way to configure the PWM for the most common operations, such as setting the period, generating left and center aligned pulses, modifying the

pulse width, and controlling interrupts, triggers, and output characteristics. However, the PWM module is very versatile, and it can be configured in a number of different ways, many of which are beyond the scope of this API. In order to fully exploit the many features of the PWM module, users are advised to use register access macros.

When discussing the various components of a PWM module, this API uses the following labeling convention:

- The three generator blocks are called **Gen0**, **Gen1**, and **Gen2**.
- The two PWM output signals associated with each generator block are called **OutA** and **OutB**.
- The six output signals are called **PWM0**, **PWM1**, **PWM2**, **PWM3**, **PWM4**, and **PWM5**.
- **PWM0** and **PWM1** are associated with **Gen0**, **PWM2** and **PWM3** are associated with **Gen1**, and **PWM4** and **PWM5** are associated with **Gen2**.

Also, as a simplifying assumption for this API, comparator A for each generator block is used exclusively to adjust the pulse width of the even numbered PWM outputs (**PWM0**, **PWM2**, and **PWM4**). In addition, comparator B is used exclusively for the odd numbered PWM outputs (**PWM1**, **PWM3**, **PWM5**).

16.2.2 Function Documentation

16.2.2.1 PWMDeadBandDisable

Disables the PWM dead band output.

Prototype:

```
void
PWMDeadBandDisable(unsigned long ulBase,
                   unsigned long ulGen)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to modify. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:

This function disables the dead band mode for the specified PWM generator. Doing so decouples the **OutA** and **OutB** signals.

Returns:

None.

16.2.2.2 PWMDeadBandEnable

Enables the PWM dead band output, and sets the dead band delays.

Prototype:

```
void
PWMDeadBandEnable(unsigned long ulBase,
```

```
                        unsigned long ulGen,
                        unsigned short usRise,
                        unsigned short usFall)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to modify. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

usRise specifies the width of delay from the rising edge.

usFall specifies the width of delay from the falling edge.

Description:

This function sets the dead bands for the specified PWM generator, where the dead bands are defined as the number of **PWM** clock ticks from the rising or falling edge of the generator's **OutA** signal. Note that this function causes the coupling of **OutB** to **OutA**.

Returns:

None.

16.2.2.3 PWMFaultIntClear

Clears the fault interrupt for a PWM module.

Prototype:

```
void
PWMFaultIntClear(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the PWM module.

Description:

Clears the fault interrupt by writing to the appropriate bit of the interrupt status register for the selected PWM module.

This function clears only the FAULT0 interrupt and is retained for backwards compatibility. It is recommended that PWMFaultIntClearExt() be used instead since it supports all fault interrupts supported on devices with and without extended PWM fault handling support.

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

16.2.2.4 PWMFaultIntClearExt

Clears the fault interrupt for a PWM module.

Prototype:

```
void
PWMFaultIntClearExt(unsigned long ulBase,
                    unsigned long ulFaultInts)
```

Parameters:

ulBase is the base address of the PWM module.

ulFaultInts specifies the fault interrupts to clear.

Description:

Clears one or more fault interrupts by writing to the appropriate bit of the PWM interrupt status register. The parameter *ulFaultInts* must be the logical OR of any of **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, or **PWM_INT_FAULT3**.

When running on a device supporting extended PWM fault handling, the fault interrupts are derived by performing a logical OR of each of the configured fault trigger signals for a given generator. Therefore, these interrupts are not directly related to the four possible FAULTn inputs to the device but indicate that a fault has been signalled to one of the four possible PWM generators. On a device without extended PWM fault handling, the interrupt is directly related to the state of the single FAULT pin.

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

16.2.2.5 PWMFaultIntRegister

Registers an interrupt handler for a fault condition detected in a PWM module.

Prototype:

```
void
PWMFaultIntRegister(unsigned long ulBase,
                    void (*pfnIntHandler)(void))
```

Parameters:

ulBase is the base address of the PWM module.

pfnIntHandler is a pointer to the function to be called when the PWM fault interrupt occurs.

Description:

This function will ensure that the interrupt handler specified by *pfnIntHandler* is called when a fault interrupt is detected for the selected PWM module. This function will also enable the

PWM fault interrupt in the NVIC; the PWM fault interrupt must also be enabled at the module level using PWMIntEnable().

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

16.2.2.6 PWMFaultIntUnregister

Removes the PWM fault condition interrupt handler.

Prototype:
```
void
PWMFaultIntUnregister(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the PWM module.

Description:
This function will remove the interrupt handler for a PWM fault interrupt from the selected PWM module. This function will also disable the PWM fault interrupt in the NVIC; the PWM fault interrupt must also be disabled at the module level using PWMIntDisable().

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

16.2.2.7 PWMGenConfigure

Configures a PWM generator.

Prototype:
```
void
PWMGenConfigure(unsigned long ulBase,
                unsigned long ulGen,
                unsigned long ulConfig)
```

Parameters:
ulBase is the base address of the PWM module.
ulGen is the PWM generator to configure. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.
ulConfig is the configuration for the PWM generator.

Description:
This function is used to set the mode of operation for a PWM generator. The counting mode, synchronization mode, and debug behavior are all configured. After configuration, the generator is left in the disabled state.

A PWM generator can count in two different modes: count down mode or count up/down mode. In count down mode, it will count from a value down to zero, and then reset to the preset value. This will produce left-aligned PWM signals (that is the rising edge of the two PWM signals produced by the generator will occur at the same time). In count up/down mode, it will count up from zero to the preset value, count back down to zero, and then repeat the process. This will produce center-aligned PWM signals (that is, the middle of the high/low period of the PWM signals produced by the generator will occur at the same time).

When the PWM generator parameters (period and pulse width) are modified, their affect on the output PWM signals can be delayed. In synchronous mode, the parameter updates are not applied until a synchronization event occurs. This allows multiple parameters to be modified and take affect simultaneously, instead of one at a time. Additionally, parameters to multiple PWM generators in synchronous mode can be updated simultaneously, allowing them to be treated as if they were a unified generator. In non-synchronous mode, the parameter updates are not delayed until a synchronization event. In either mode, the parameter updates only occur when the counter is at zero to help prevent oddly formed PWM signals during the update (that is, a PWM pulse that is too short or too long).

The PWM generator can either pause or continue running when the processor is stopped via the debugger. If configured to pause, it will continue to count until it reaches zero, at which point it will pause until the processor is restarted. If configured to continue running, it will keep counting as if nothing had happened.

The *ulConfig* parameter contains the desired configuration. It is the logical OR of the following:

- **PWM_GEN_MODE_DOWN** or **PWM_GEN_MODE_UP_DOWN** to specify the counting mode
- **PWM_GEN_MODE_SYNC** or **PWM_GEN_MODE_NO_SYNC** to specify the counter load and comparator update synchronization mode
- **PWM_GEN_MODE_DBG_RUN** or **PWM_GEN_MODE_DBG_STOP** to specify the debug behavior
- **PWM_GEN_MODE_GEN_NO_SYNC**, **PWM_GEN_MODE_GEN_SYNC_LOCAL**, or **PWM_GEN_MODE_GEN_SYNC_GLOBAL** to specify the update synchronization mode for generator counting mode changes
- **PWM_GEN_MODE_DB_NO_SYNC**, **PWM_GEN_MODE_DB_SYNC_LOCAL**, or **PWM_GEN_MODE_DB_SYNC_GLOBAL** to specify the deadband parameter synchronization mode
- **PWM_GEN_MODE_FAULT_LATCHED** or **PWM_GEN_MODE_FAULT_UNLATCHED** to specify whether fault conditions are latched or not
- **PWM_GEN_MODE_FAULT_MINPER** or **PWM_GEN_MODE_FAULT_NO_MINPER** to specify whether minimum fault period support is required
- **PWM_GEN_MODE_FAULT_EXT** or **PWM_GEN_MODE_FAULT_LEGACY** to specify whether extended fault source selection support is enabled or not

Setting **PWM_GEN_MODE_FAULT_MINPER** allows an application to set the minimum duration of a PWM fault signal. Fault will be signalled for at least this time even if the external fault pin deasserts earlier. Care should be taken when using this mode since during the fault signal period, the fault interrupt from the PWM generator will remain asserted. The fault interrupt handler may, therefore, reenter immediately if it exits prior to expiration of the fault timer.

Note:
Changes to the counter mode will affect the period of the PWM signals produced. PWMGenPeriodSet() and PWMPulseWidthSet() should be called after any changes to the counter mode of a generator.

Returns:
None.

16.2.2.8 PWMGenDisable

Disables the timer/counter for a PWM generator block.

Prototype:
```
void
PWMGenDisable(unsigned long ulBase,
              unsigned long ulGen)
```

Parameters:
ulBase is the base address of the PWM module.
ulGen is the PWM generator to be disabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:
This function blocks the PWM clock from driving the timer/counter for the specified generator block.

Returns:
None.

16.2.2.9 PWMGenEnable

Enables the timer/counter for a PWM generator block.

Prototype:
```
void
PWMGenEnable(unsigned long ulBase,
             unsigned long ulGen)
```

Parameters:
ulBase is the base address of the PWM module.
ulGen is the PWM generator to be enabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:
This function allows the PWM clock to drive the timer/counter for the specified generator block.

Returns:
None.

16.2.2.10 PWMGenFaultClear

Clears one or more latched fault triggers for a given PWM generator.

Prototype:

```
void
PWMGenFaultClear(unsigned long ulBase,
                 unsigned long ulGen,
                 unsigned long ulGroup,
                 unsigned long ulFaultTriggers)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator whose fault trigger states are being queried. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulGroup indicates the subset of faults that are being queried. This must be **PWM_FAULT_GROUP_0**.

ulFaultTriggers is the set of fault triggers which are to be cleared.

Description:

This function allows an application to clear the fault triggers for a given PWM generator. This is only required if PWMGenConfigure() has previously been called with flag **PWM_GEN_MODE_LATCH_FAULT** in parameter *ulConfig*.

Note:

This function is only available on devices supporting extended PWM fault handling.

Returns:

None.

16.2.2.11 PWMGenFaultConfigure

Configures the minimum fault period and fault pin senses for a given PWM generator.

Prototype:

```
void
PWMGenFaultConfigure(unsigned long ulBase,
                     unsigned long ulGen,
                     unsigned long ulMinFaultPeriod,
                     unsigned long ulFaultSenses)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator whose fault configuration is being set. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulMinFaultPeriod is the minimum fault active period expressed in PWM clock cycles.

ulFaultSenses indicates which sense of each FAULT input should be considered the "asserted" state. Valid values are logical OR combinations of **PWM_FAULTn_SENSE_HIGH** and **PWM_FAULTn_SENSE_LOW**.

Description:

This function sets the minimum fault period for a given generator along with the sense of each of the 4 possible fault inputs. The minimum fault period is expressed in PWM clock cycles and takes effect only if PWMGenConfigure() is called with flag **PWM_GEN_MODE_FAULT_PER** set in the *ulConfig* parameter. When a fault input is asserted, the minimum fault period timer ensures that it remains asserted for at least the number of clock cycles specified.

Note:
This function is only available on devices supporting extended PWM fault handling.

Returns:
None.

16.2.2.12 PWMGenFaultStatus

Returns the current state of the fault triggers for a given PWM generator.

Prototype:
```
unsigned long
PWMGenFaultStatus(unsigned long ulBase,
                  unsigned long ulGen,
                  unsigned long ulGroup)
```

Parameters:
ulBase is the base address of the PWM module.

ulGen is the PWM generator whose fault trigger states are being queried. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulGroup indicates the subset of faults that are being queried. This must be **PWM_FAULT_GROUP_0**.

Description:
This function allows an application to query the current state of each of the fault trigger inputs to a given PWM generator. The current state of each fault trigger input is returned unless PWMGenConfigure() has previously been called with flag **PWM_GEN_MODE_LATCH_FAULT** in the *ulConfig* parameter in which case the returned status is the latched fault trigger status.

If latched faults are configured, the application must call PWMGenFaultClear() to clear each trigger.

Note:
This function is only available on devices supporting extended PWM fault handling.

Returns:
Returns the current state of the fault triggers for the given PWM generator. A set bit indicates that the associated trigger is active. For **PWM_FAULT_GROUP_0**, the returned value will be a logical OR of **PWM_FAULT_FAULT0**, **PWM_FAULT_FAULT1**, **PWM_FAULT_FAULT2**, or **PWM_FAULT_FAULT3**.

16.2.2.13 PWMGenFaultTriggerGet

Returns the set of fault triggers currently configured for a given PWM generator.

Prototype:
```
unsigned long
PWMGenFaultTriggerGet(unsigned long ulBase,
                      unsigned long ulGen,
                      unsigned long ulGroup)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator whose fault triggers are being queried. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulGroup indicates the subset of faults that are being queried. This must be **PWM_FAULT_GROUP_0**.

Description:

This function allows an application to query the current set of inputs that contribute towards the generation of a fault condition to a given PWM generator.

Note:

This function is only available on devices supporting extended PWM fault handling.

Returns:

Returns the current fault triggers configured for the fault group provided. For **PWM_FAULT_GROUP_0**, the returned value will be a logical OR of **PWM_FAULT_FAULT0**, **PWM_FAULT_FAULT1**, **PWM_FAULT_FAULT2**, or **PWM_FAULT_FAULT3**.

16.2.2.14 PWMGenFaultTriggerSet

Configures the set of fault triggers for a given PWM generator.

Prototype:

```
void
PWMGenFaultTriggerSet(unsigned long ulBase,
                      unsigned long ulGen,
                      unsigned long ulGroup,
                      unsigned long ulFaultTriggers)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator whose fault triggers are being set. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulGroup indicates the subset of possible faults that are to be configured. This must be **PWM_FAULT_GROUP_0**.

ulFaultTriggers defines the set of inputs that are to contribute towards generation of the fault signal to the given PWM generator. For **PWM_FAULT_GROUP_0**, this will be the logical OR of **PWM_FAULT_FAULT0**, **PWM_FAULT_FAULT1**, **PWM_FAULT_FAULT2**, or **PWM_FAULT_FAULT3**.

Description:

This function allows selection of the set of fault inputs that will be combined to generate a fault condition to a given PWM generator. By default, all generators use only FAULT0 (for backwards compatibility) but if PWMGenConfigure() is called with flag **PWM_GEN_MODE_FAULT_SRC** in the *ulConfig* parameter, extended fault handling is enabled and this function must be called to configure the fault triggers.

The fault signal to the PWM generator is generated by ORing together each of the signals whose inputs are specified in the *ulFaultTriggers* parameter after having adjusted the sense of each FAULTn input based on the configuration previously set using a call to PWMGenFaultConfigure().

Note:
This function is only available on devices supporting extended PWM fault handling.

Returns:
None.

16.2.2.15 PWMGenIntClear

Clears the specified interrupt(s) for the specified PWM generator block.

Prototype:
```
void
PWMGenIntClear(unsigned long ulBase,
               unsigned long ulGen,
               unsigned long ulInts)
```

Parameters:
ulBase is the base address of the PWM module.
ulGen is the PWM generator to query. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.
ulInts specifies the interrupts to be cleared.

Description:
Clears the specified interrupt(s) by writing a 1 to the specified bits of the interrupt status register for the specified PWM generator. The *ulInts* parameter is the logical OR of **PWM_INT_CNT_ZERO**, **PWM_INT_CNT_LOAD**, **PWM_INT_CNT_AU**, **PWM_INT_CNT_AD**, **PWM_INT_CNT_BU**, or **PWM_INT_CNT_BD**.

Note:
Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:
None.

16.2.2.16 PWMGenIntRegister

Registers an interrupt handler for the specified PWM generator block.

Prototype:
```
void
PWMGenIntRegister(unsigned long ulBase,
                  unsigned long ulGen,
                  void (*pfnIntHandler)(void))
```

Parameters:
ulBase is the base address of the PWM module.

ulGen is the PWM generator in question. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

pfnIntHandler is a pointer to the function to be called when the PWM generator interrupt occurs.

Description:

This function will ensure that the interrupt handler specified by *pfnIntHandler* is called when an interrupt is detected for the specified PWM generator block. This function will also enable the corresponding PWM generator interrupt in the interrupt controller; individual generator interrupts and interrupt sources must be enabled with PWMIntEnable() and PWMGenIntTrigEnable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

16.2.2.17 PWMGenIntStatus

Gets interrupt status for the specified PWM generator block.

Prototype:

```
unsigned long
PWMGenIntStatus(unsigned long ulBase,
                unsigned long ulGen,
                tBoolean bMasked)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to query. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

bMasked specifies whether masked or raw interrupt status is returned.

Description:

If *bMasked* is set as **true**, then the masked interrupt status is returned; otherwise, the raw interrupt status will be returned.

Returns:

Returns the contents of the interrupt status register, or the contents of the raw interrupt status register, for the specified PWM generator.

16.2.2.18 PWMGenIntTrigDisable

Disables interrupts for the specified PWM generator block.

Prototype:

```
void
PWMGenIntTrigDisable(unsigned long ulBase,
                     unsigned long ulGen,
                     unsigned long ulIntTrig)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to have interrupts and triggers disabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulIntTrig specifies the interrupts and triggers to be disabled.

Description:

Masks the specified interrupt(s) and trigger(s) by clearing the specified bits of the interrupt/trigger enable register for the specified PWM generator. The *ulIntTrig* parameter is the logical OR of **PWM_INT_CNT_ZERO**, **PWM_INT_CNT_LOAD**, **PWM_INT_CNT_AU**, **PWM_INT_CNT_AD**, **PWM_INT_CNT_BU**, **PWM_INT_CNT_BD**, **PWM_TR_CNT_ZERO**, **PWM_TR_CNT_LOAD**, **PWM_TR_CNT_AU**, **PWM_TR_CNT_AD**, **PWM_TR_CNT_BU**, or **PWM_TR_CNT_BD**.

Returns:

None.

16.2.2.19 PWMGenIntTrigEnable

Enables interrupts and triggers for the specified PWM generator block.

Prototype:

```
void
PWMGenIntTrigEnable(unsigned long ulBase,
                    unsigned long ulGen,
                    unsigned long ulIntTrig)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to have interrupts and triggers enabled. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulIntTrig specifies the interrupts and triggers to be enabled.

Description:

Unmasks the specified interrupt(s) and trigger(s) by setting the specified bits of the interrupt/trigger enable register for the specified PWM generator. The *ulIntTrig* parameter is the logical OR of **PWM_INT_CNT_ZERO**, **PWM_INT_CNT_LOAD**, **PWM_INT_CNT_AU**, **PWM_INT_CNT_AD**, **PWM_INT_CNT_BU**, **PWM_INT_CNT_BD**, **PWM_TR_CNT_ZERO**, **PWM_TR_CNT_LOAD**, **PWM_TR_CNT_AU**, **PWM_TR_CNT_AD**, **PWM_TR_CNT_BU**, or **PWM_TR_CNT_BD**.

Returns:

None.

16.2.2.20 PWMGenIntUnregister

Removes an interrupt handler for the specified PWM generator block.

Prototype:

```
void
PWMGenIntUnregister(unsigned long ulBase,
                    unsigned long ulGen)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator in question. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:

This function will unregister the interrupt handler for the specified PWM generator block. This function will also disable the corresponding PWM generator interrupt in the interrupt controller; individual generator interrupts and interrupt sources must be disabled with PWMIntDisable() and PWMGenIntTrigDisable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

16.2.2.21 PWMGenPeriodGet

Gets the period of a PWM generator block.

Prototype:

```
unsigned long
PWMGenPeriodGet(unsigned long ulBase,
                unsigned long ulGen)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to query. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

Description:

This function gets the period of the specified PWM generator block. The period of the generator block is defined as the number of PWM clock ticks between pulses on the generator block zero signal.

If the update of the counter for the specified PWM generator has yet to be completed, the value returned may not be the active period. The value returned is the programmed period, measured in PWM clock ticks.

Returns:

Returns the programmed period of the specified generator block in PWM clock ticks.

16.2.2.22 PWMGenPeriodSet

Set the period of a PWM generator.

Prototype:

```
void
PWMGenPeriodSet(unsigned long ulBase,
                unsigned long ulGen,
                unsigned long ulPeriod)
```

Parameters:

ulBase is the base address of the PWM module.

ulGen is the PWM generator to be modified. Must be one of **PWM_GEN_0**, **PWM_GEN_1**, **PWM_GEN_2**, or **PWM_GEN_3**.

ulPeriod specifies the period of PWM generator output, measured in clock ticks.

Description:

This function sets the period of the specified PWM generator block, where the period of the generator block is defined as the number of PWM clock ticks between pulses on the generator block zero signal.

Note:

Any subsequent calls made to this function before an update occurs will cause the previous values to be overwritten.

Returns:

None.

16.2.2.23 PWMIntDisable

Disables generator and fault interrupts for a PWM module.

Prototype:

```
void
PWMIntDisable(unsigned long ulBase,
              unsigned long ulGenFault)
```

Parameters:

ulBase is the base address of the PWM module.

ulGenFault contains the interrupts to be disabled. Must be a logical OR of any of **PWM_INT_GEN_0**, **PWM_INT_GEN_1**, **PWM_INT_GEN_2**, **PWM_INT_GEN_3**, **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, or **PWM_INT_FAULT3**.

Description:

Masks the specified interrupt(s) by clearing the specified bits of the interrupt enable register for the selected PWM module.

Returns:

None.

16.2.2.24 PWMIntEnable

Enables generator and fault interrupts for a PWM module.

Prototype:

```
void
PWMIntEnable(unsigned long ulBase,
             unsigned long ulGenFault)
```

Parameters:

ulBase is the base address of the PWM module.

ulGenFault contains the interrupts to be enabled. Must be a logical OR of any of **PWM_INT_GEN_0**, **PWM_INT_GEN_1**, **PWM_INT_GEN_2**, **PWM_INT_GEN_3**, **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, or **PWM_INT_FAULT3**.

Description:

Unmasks the specified interrupt(s) by setting the specified bits of the interrupt enable register for the selected PWM module.

Returns:

None.

16.2.2.25 PWMIntStatus

Gets the interrupt status for a PWM module.

Prototype:

```
unsigned long
PWMIntStatus(unsigned long ulBase,
             tBoolean bMasked)
```

Parameters:

ulBase is the base address of the PWM module.

bMasked specifies whether masked or raw interrupt status is returned.

Description:

If *bMasked* is set as **true**, then the masked interrupt status is returned; otherwise, the raw interrupt status will be returned.

Returns:

The current interrupt status, enumerated as a bit field of **PWM_INT_GEN_0**, **PWM_INT_GEN_1**, **PWM_INT_GEN_2**, **PWM_INT_GEN_3**, **PWM_INT_FAULT0**, **PWM_INT_FAULT1**, **PWM_INT_FAULT2**, and **PWM_INT_FAULT3**.

16.2.2.26 PWMOutputFault

Specifies the state of PWM outputs in response to a fault condition.

Prototype:

```
void
PWMOutputFault(unsigned long ulBase,
               unsigned long ulPWMOutBits,
               tBoolean bFaultSuppress)
```

Parameters:

ulBase is the base address of the PWM module.

ulPWMOutBits are the PWM outputs to be modified. Must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

bFaultSuppress determines if the signal is suppressed or passed through during an active fault condition.

Description:

This function sets the fault handling characteristics of the selected PWM outputs. The outputs are selected using the parameter *ulPWMOutBits*. The parameter *bFaultSuppress* determines the fault handling characteristics for the selected outputs. If *bFaultSuppress* is **true**, then the selected outputs will be made inactive. If *bFaultSuppress* is **false**, then the selected outputs are unaffected by the detected fault.

On devices supporting extended PWM fault handling, the state the affected output pins are driven to can be configured with PWMOutputFaultLevel(). If not configured, or if the device does not support extended PWM fault handling, affected outputs will be driven low on a fault condition.

Returns:

None.

16.2.2.27 PWMOutputFaultLevel

Specifies the level of PWM outputs suppressed in response to a fault condition.

Prototype:

```
void
PWMOutputFaultLevel(unsigned long ulBase,
                    unsigned long ulPWMOutBits,
                    tBoolean bDriveHigh)
```

Parameters:

ulBase is the base address of the PWM module.

ulPWMOutBits are the PWM outputs to be modified. Must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

bDriveHigh determines if the signal is driven high or low during an active fault condition.

Description:

This function determines whether a PWM output pin that is suppressed in response to a fault condition will be driven high or low. The affected outputs are selected using the parameter *ulPWMOutBits*. The parameter *bDriveHigh* determines the output level for the pins identified by *ulPWMOutBits*. If *bDriveHigh* is **true** then the selected outputs will be driven high when a fault is detected. If it is *false*, the pins will be driven low.

In a fault condition, pins which have not been configured to be suppressed via a call to PWMOutputFault() are unaffected by this function.

Note:

This function is available only on devices which support extended PWM fault handling.

Returns:

None.

16.2.2.28 PWMOutputInvert

Selects the inversion mode for PWM outputs.

Prototype:

```
void
PWMOutputInvert(unsigned long ulBase,
                unsigned long ulPWMOutBits,
                tBoolean bInvert)
```

Parameters:

ulBase is the base address of the PWM module.

ulPWMOutBits are the PWM outputs to be modified. Must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

bInvert determines if the signal is inverted or passed through.

Description:

This function is used to select the inversion mode for the selected PWM outputs. The outputs are selected using the parameter *ulPWMOutBits*. The parameter *bInvert* determines the inversion mode for the selected outputs. If *bInvert* is **true**, this function will cause the specified PWM output signals to be inverted, or made active low. If *bInvert* is **false**, the specified output will be passed through as is, or be made active high.

Returns:

None.

16.2.2.29 PWMOutputState

Enables or disables PWM outputs.

Prototype:

```
void
PWMOutputState(unsigned long ulBase,
               unsigned long ulPWMOutBits,
               tBoolean bEnable)
```

Parameters:

ulBase is the base address of the PWM module.

ulPWMOutBits are the PWM outputs to be modified. Must be the logical OR of any of **PWM_OUT_0_BIT**, **PWM_OUT_1_BIT**, **PWM_OUT_2_BIT**, **PWM_OUT_3_BIT**, **PWM_OUT_4_BIT**, **PWM_OUT_5_BIT**, **PWM_OUT_6_BIT**, or **PWM_OUT_7_BIT**.

bEnable determines if the signal is enabled or disabled.

Description:

This function is used to enable or disable the selected PWM outputs. The outputs are selected using the parameter *ulPWMOutBits*. The parameter *bEnable* determines the state of the selected outputs. If *bEnable* is **true**, then the selected PWM outputs are enabled, or placed in the active state. If *bEnable* is **false**, then the selected outputs are disabled, or placed in the inactive state.

Returns:

None.

16.2.2.30 PWMPulseWidthGet

Gets the pulse width of a PWM output.

Prototype:

```
unsigned long
PWMPulseWidthGet(unsigned long ulBase,
                 unsigned long ulPWMOut)
```

Parameters:

ulBase is the base address of the PWM module.

ulPWMOut is the PWM output to query. Must be one of **PWM_OUT_0**, **PWM_OUT_1**, **PWM_OUT_2**, **PWM_OUT_3**, **PWM_OUT_4**, **PWM_OUT_5**, **PWM_OUT_6**, or **PWM_OUT_7**.

Description:

This function gets the currently programmed pulse width for the specified PWM output. If the update of the comparator for the specified output has yet to be completed, the value returned may not be the active pulse width. The value returned is the programmed pulse width, measured in PWM clock ticks.

Returns:

Returns the width of the pulse in PWM clock ticks.

16.2.2.31 PWMPulseWidthSet

Sets the pulse width for the specified PWM output.

Prototype:

```
void
PWMPulseWidthSet(unsigned long ulBase,
                 unsigned long ulPWMOut,
                 unsigned long ulWidth)
```

Parameters:

ulBase is the base address of the PWM module.

ulPWMOut is the PWM output to modify. Must be one of **PWM_OUT_0**, **PWM_OUT_1**, **PWM_OUT_2**, **PWM_OUT_3**, **PWM_OUT_4**, **PWM_OUT_5**, **PWM_OUT_6**, or **PWM_OUT_7**.

ulWidth specifies the width of the positive portion of the pulse.

Description:

This function sets the pulse width for the specified PWM output, where the pulse width is defined as the number of PWM clock ticks.

Note:

Any subsequent calls made to this function before an update occurs will cause the previous values to be overwritten.

Returns:

None.

16.2.2.32 PWMSyncTimeBase

Synchronizes the counters in one or multiple PWM generator blocks.

Prototype:

```
void
PWMSyncTimeBase(unsigned long ulBase,
                unsigned long ulGenBits)
```

Parameters:

ulBase is the base address of the PWM module.

ulGenBits are the PWM generator blocks to be synchronized. Must be the logical OR of any of **PWM_GEN_0_BIT**, **PWM_GEN_1_BIT**, **PWM_GEN_2_BIT**, or **PWM_GEN_3_BIT**.

Description:

For the selected PWM module, this function synchronizes the time base of the generator blocks by causing the specified generator counters to be reset to zero.

Returns:

None.

16.2.2.33 PWMSyncUpdate

Synchronizes all pending updates.

Prototype:

```
void
PWMSyncUpdate(unsigned long ulBase,
              unsigned long ulGenBits)
```

Parameters:

ulBase is the base address of the PWM module.

ulGenBits are the PWM generator blocks to be updated. Must be the logical OR of any of **PWM_GEN_0_BIT**, **PWM_GEN_1_BIT**, **PWM_GEN_2_BIT**, or **PWM_GEN_3_BIT**.

Description:

For the selected PWM generators, this function causes all queued updates to the period or pulse width to be applied the next time the corresponding counter becomes zero.

Returns:

None.

16.3 Programming Example

The following example shows how to use the PWM API to initialize the PWM0 with a 50 KHz frequency, and with a 25% duty cycle on **PWM0** and a 75% duty cycle on **PWM1**.

```
//
// Configure the PWM generator for count down mode with immediate updates
// to the parameters.
```

```
//
PWMGenConfigure(PWM_BASE, PWM_GEN_0,
                PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);

//
// Set the period.  For a 50 KHz frequency, the period = 1/50,000, or 20
// microseconds.  For a 20 MHz clock, this translates to 400 clock ticks.
// Use this value to set the period.
//
PWMGenPeriodSet(PWM_BASE, PWM_GEN_0, 400);

//
// Set the pulse width of PWM0 for a 25% duty cycle.
//
PWMPulseWidthSet(PWM_BASE, PWM_OUT_0, 100);

//
// Set the pulse width of PWM1 for a 75% duty cycle.
//
PWMPulseWidthSet(PWM_BASE, PWM_OUT_1, 300);

//
// Start the timers in generator 0.
//
PWMGenEnable(PWM_BASE, PWM_GEN_0);

//
// Enable the outputs.
//
PWMOutputState(PWM_BASE, (PWM_OUT_0_BIT | PWM_OUT_1_BIT), true);
```

17 Quadrature Encoder (QEI)

17.1 Introduction

The quadrature encoder API provides a set of functions for dealing with the Quadrature Encoder with Index (QEI). Functions are provided to configure and read the position and velocity captures, register a QEI interrupt handler, and handle QEI interrupt masking/clearing.

The quadrature encoder module provides hardware encoding of the two channels and the index signal from a quadrature encoder device into an absolute or relative position. There is additional hardware for capturing a measure of the encoder velocity, which is simply a count of encoder pulses during a fixed time period; the number of pulses is directly proportional to the encoder speed. Note that the velocity capture can only operate when the position capture is enabled.

The QEI module supports two modes of operation: phase mode and clock/direction mode. In phase mode, the encoder produces two clocks that are 90 degrees out of phase; the edge relationship is used to determine the direction of rotation. In clock/direction mode, the encoder produces a clock signal to indicate steps and a direction signal to indicate the direction of rotation.

When in phase mode, edges on the first channel or edges on both channels can be counted; counting edges on both channels provides higher encoder resolution if required. In either mode, the input signals can be swapped before being processed; this allows wiring mistakes on the circuit board to be corrected without modifying the board.

The index pulse can be used to reset the position counter; this causes the position counter to maintain the absolute encoder position. Otherwise, the position counter maintains the relative position and is never reset.

The velocity capture has a timer to measure equal periods of time. The number of encoder pulses over each time period is accumulated as a measure of the encoder velocity. The running total for the current time period and the final count for the previous time period are available to be read. The final count for the previous time period is usually used as the velocity measure.

The QEI module will generate interrupts when the index pulse is detected, when the velocity timer expires, when the encoder direction changes, and when a phase signal error is detected. These interrupt sources can be individually masked so that only the events of interest cause a processor interrupt.

This driver is contained in `src/qei.c`, with `src/qei.h` containing the API definitions for use by applications.

17.2 API Functions

Functions

- void QEIConfigure (unsigned long ulBase, unsigned long ulConfig, unsigned long ulMaxPosition)
- long QEIDirectionGet (unsigned long ulBase)
- void QEIDisable (unsigned long ulBase)
- void QEIEnable (unsigned long ulBase)
- tBoolean QEIErrorGet (unsigned long ulBase)
- void QEIIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void QEIIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void QEIIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- void QEIIntRegister (unsigned long ulBase, void (*pfnHandler)(void))
- unsigned long QEIIntStatus (unsigned long ulBase, tBoolean bMasked)
- void QEIIntUnregister (unsigned long ulBase)
- unsigned long QEIPositionGet (unsigned long ulBase)
- void QEIPositionSet (unsigned long ulBase, unsigned long ulPosition)
- void QEIVelocityConfigure (unsigned long ulBase, unsigned long ulPreDiv, unsigned long ulPeriod)
- void QEIVelocityDisable (unsigned long ulBase)
- void QEIVelocityEnable (unsigned long ulBase)
- unsigned long QEIVelocityGet (unsigned long ulBase)

17.2.1 Detailed Description

The quadrature encoder API is broken into three groups of functions: those that deal with position capture, those that deal with velocity capture, and those that deal with interrupt handling.

The position capture is managed with QEIEnable(), QEIDisable(),QEIConfigure(), and QEIPositionSet(). The positional information is retrieved with QEIPositionGet(), QEIDirectionGet(), and QEIErrorGet().

The velocity capture is managed with QEIVelocityEnable(), QEIVelocityDisable(), and QEIVelocityConfigure(). The computed encoder velocity is retrieved with QEIVelocityGet().

The interrupt handler for the QEI interrupt is managed with QEIIntRegister() and QEIIntUnregister(). The individual interrupt sources within the QEI module are managed with QEIIntEnable(), QEIIntDisable(), QEIIntStatus(), and QEIIntClear().

17.2.2 Function Documentation

17.2.2.1 QEIConfigure

Configures the quadrature encoder.

Prototype:

```
void
QEIConfigure(unsigned long ulBase,
             unsigned long ulConfig,
             unsigned long ulMaxPosition)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

ulConfig is the configuration for the quadrature encoder. See below for a description of this parameter.

ulMaxPosition specifies the maximum position value.

Description:

This will configure the operation of the quadrature encoder. The *ulConfig* parameter provides the configuration of the encoder and is the logical OR of several values:

- **QEI_CONFIG_CAPTURE_A** or **QEI_CONFIG_CAPTURE_A_B** to specify if edges on channel A or on both channels A and B should be counted by the position integrator and velocity accumulator.
- **QEI_CONFIG_NO_RESET** or **QEI_CONFIG_RESET_IDX** to specify if the position integrator should be reset when the index pulse is detected.
- **QEI_CONFIG_QUADRATURE** or **QEI_CONFIG_CLOCK_DIR** to specify if quadrature signals are being provided on ChA and ChB, or if a direction signal and a clock are being provided instead.
- **QEI_CONFIG_NO_SWAP** or **QEI_CONFIG_SWAP** to specify if the signals provided on ChA and ChB should be swapped before being processed.

ulMaxPosition is the maximum value of the position integrator, and is the value used to reset the position capture when in index reset mode and moving in the reverse (negative) direction.

Returns:

None.

17.2.2.2 QEIDirectionGet

Gets the current direction of rotation.

Prototype:

```
long
QEIDirectionGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This returns the current direction of rotation. In this case, current means the most recently detected direction of the encoder; it may not be presently moving but this is the direction it last moved before it stopped.

Returns:

Returns 1 if moving in the forward direction or -1 if moving in the reverse direction.

17.2.2.3 QEIDisable

Disables the quadrature encoder.

Prototype:

```
void
QEIDisable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This will disable operation of the quadrature encoder module.

Returns:

None.

17.2.2.4 QEIEnable

Enables the quadrature encoder.

Prototype:

```
void
QEIEnable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This will enable operation of the quadrature encoder module. It must be configured before it is enabled.

See also:

QEIConfigure()

Returns:

None.

17.2.2.5 QEIErrorGet

Gets the encoder error indicator.

Prototype:

```
tBoolean
QEIErrorGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This returns the error indicator for the quadrature encoder. It is an error for both of the signals of the quadrature input to change at the same time.

Returns:
Returns **true** if an error has occurred and **false** otherwise.

17.2.2.6 QEIIntClear

Clears quadrature encoder interrupt sources.

Prototype:
```
void
QEIIntClear(unsigned long ulBase,
            unsigned long ulIntFlags)
```

Parameters:
ulBase is the base address of the quadrature encoder module.

ulIntFlags is a bit mask of the interrupt sources to be cleared. Can be any of the **QEI_INTERROR**, **QEI_INTDIR**, **QEI_INTTIMER**, or **QEI_INTINDEX** values.

Description:
The specified quadrature encoder interrupt sources are cleared, so that they no longer assert. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

Note:
Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:
None.

17.2.2.7 QEIIntDisable

Disables individual quadrature encoder interrupt sources.

Prototype:
```
void
QEIIntDisable(unsigned long ulBase,
              unsigned long ulIntFlags)
```

Parameters:
ulBase is the base address of the quadrature encoder module.

ulIntFlags is a bit mask of the interrupt sources to be disabled. Can be any of the **QEI_INTERROR**, **QEI_INTDIR**, **QEI_INTTIMER**, or **QEI_INTINDEX** values.

Description:
Disables the indicated quadrature encoder interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns:
None.

17.2.2.8 QEIIntEnable

Enables individual quadrature encoder interrupt sources.

Prototype:
```
void
QEIIntEnable(unsigned long ulBase,
             unsigned long ulIntFlags)
```

Parameters:
ulBase is the base address of the quadrature encoder module.

ulIntFlags is a bit mask of the interrupt sources to be enabled. Can be any of the **QEI_INTERROR**, **QEI_INTDIR**, **QEI_INTTIMER**, or **QEI_INTINDEX** values.

Description:
Enables the indicated quadrature encoder interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns:
None.

17.2.2.9 QEIIntRegister

Registers an interrupt handler for the quadrature encoder interrupt.

Prototype:
```
void
QEIIntRegister(unsigned long ulBase,
               void (*pfnHandler)(void))
```

Parameters:
ulBase is the base address of the quadrature encoder module.

pfnHandler is a pointer to the function to be called when the quadrature encoder interrupt occurs.

Description:
This sets the handler to be called when a quadrature encoder interrupt occurs. This will enable the global interrupt in the interrupt controller; specific quadrature encoder interrupts must be enabled via QEIIntEnable(). It is the interrupt handler's responsibility to clear the interrupt source via QEIIntClear().

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

17.2.2.10 QEIIntStatus

Gets the current interrupt status.

Prototype:

```
unsigned long
QEIIntStatus(unsigned long ulBase,
             tBoolean bMasked)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This returns the interrupt status for the quadrature encoder module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status, enumerated as a bit field of **QEI_INTERROR**, **QEI_INTDIR**, **QEI_INTTIMER**, and **QEI_INTINDEX**.

17.2.2.11 QEIIntUnregister

Unregisters an interrupt handler for the quadrature encoder interrupt.

Prototype:

```
void
QEIIntUnregister(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This function will clear the handler to be called when a quadrature encoder interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

17.2.2.12 QEIPositionGet

Gets the current encoder position.

Prototype:

```
unsigned long
QEIPositionGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This returns the current position of the encoder. Depending upon the configuration of the encoder, and the incident of an index pulse, this value may or may not contain the expected data (that is, if in reset on index mode, if an index pulse has not been encountered, the position counter will not be aligned with the index pulse yet).

Returns:

The current position of the encoder.

17.2.2.13 QEIPositionSet

Sets the current encoder position.

Prototype:

```
void
QEIPositionSet(unsigned long ulBase,
               unsigned long ulPosition)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

ulPosition is the new position for the encoder.

Description:

This sets the current position of the encoder; the encoder position will then be measured relative to this value.

Returns:

None.

17.2.2.14 QEIVelocityConfigure

Configures the velocity capture.

Prototype:

```
void
QEIVelocityConfigure(unsigned long ulBase,
                     unsigned long ulPreDiv,
                     unsigned long ulPeriod)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

ulPreDiv specifies the predivider applied to the input quadrature signal before it is counted; can be one of **QEI_VELDIV_1**, **QEI_VELDIV_2**, **QEI_VELDIV_4**, **QEI_VELDIV_8**, **QEI_VELDIV_16**, **QEI_VELDIV_32**, **QEI_VELDIV_64**, or **QEI_VELDIV_128**.

ulPeriod specifies the number of clock ticks over which to measure the velocity; must be non-zero.

Description:
This will configure the operation of the velocity capture portion of the quadrature encoder. The position increment signal is predivided as specified by *ulPreDiv* before being accumulated by the velocity capture. The divided signal is accumulated over *ulPeriod* system clock before being saved and resetting the accumulator.

Returns:
None.

17.2.2.15 QEIVelocityDisable

Disables the velocity capture.

Prototype:
```
void
QEIVelocityDisable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the quadrature encoder module.

Description:
This will disable operation of the velocity capture in the quadrature encoder module.

Returns:
None.

17.2.2.16 QEIVelocityEnable

Enables the velocity capture.

Prototype:
```
void
QEIVelocityEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the quadrature encoder module.

Description:
This will enable operation of the velocity capture in the quadrature encoder module. It must be configured before it is enabled. Velocity capture will not occur if the quadrature encoder is not enabled.

See also:
QEIVelocityConfigure() and QEIEnable()

Returns:
None.

17.2.2.17 QEIVelocityGet

Gets the current encoder speed.

Prototype:

```
unsigned long
QEIVelocityGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the quadrature encoder module.

Description:

This returns the current speed of the encoder. The value returned is the number of pulses detected in the specified time period; this number can be multiplied by the number of time periods per second and divided by the number of pulses per revolution to obtain the number of revolutions per second.

Returns:

Returns the number of pulses captured in the given time period.

17.3 Programming Example

The following example shows how to use the Quadrature Encoder API to configure the quadrature encoder read back an absolute position.

```
//
// Configure the quadrature encoder to capture edges on both signals and
// maintain an absolute position by resetting on index pulses.  Using a
// 1000 line encoder at four edges per line, there are 4000 pulses per
// revolution; therefore set the maximum position to 3999 since the count
// is zero based.
//
QEIConfigure(QEI_BASE, (QEI_CONFIG_CAPTURE_A_B | QEI_CONFIG_RESET_IDX |
                        QEI_CONFIG_QUADRATURE | QEI_CONFIG_NO_SWAP), 3999);

//
// Enable the quadrature encoder.
//
QEIEnable(QEI_BASE);

//
// Delay for some time...
//

//
// Read the encoder position.
//
QEIPositionGet(QEI_BASE);
```

18 Synchronous Serial Interface (SSI)

18.1 Introduction

The Synchronous Serial Interface (SSI) module provides the functionality for synchronous serial communications with peripheral devices, and can be configured to use either the Motorola® SPI™, National Semiconductor® Microwire, or the Texas Instruments® synchronous serial interface frame formats. The size of the data frame is also configurable, and can be set to be between 4 and 16 bits, inclusive.

The SSI module performs serial-to-parallel data conversion on data received from a peripheral device, and parallel-to-serial conversion on data transmitted to a peripheral device. The TX and RX paths are buffered with internal FIFOs allowing up to eight 16-bit values to be stored independently.

The SSI module can be configured as either a master or a slave device. As a slave device, the SSI module can also be configured to disable its output, which allows a master device to be coupled with multiple slave devices.

The SSI module also includes a programmable bit rate clock divider and prescaler to generate the output serial clock derived from the SSI module's input clock. Bit rates are generated based on the input clock and the maximum bit rate supported by the connected peripheral.

For parts that include a DMA controller, the SSI module also provides a DMA interface to facilitate data transfer via DMA.

This driver is contained in `src/ssi.c`, with `src/ssi.h` containing the API definitions for use by applications.

18.2 API Functions

Functions

- void SSIConfigSetExpClk (unsigned long ulBase, unsigned long ulSSIClk, unsigned long ulProtocol, unsigned long ulMode, unsigned long ulBitRate, unsigned long ulDataWidth)
- void SSIDataGet (unsigned long ulBase, unsigned long *pulData)
- long SSIDataGetNonBlocking (unsigned long ulBase, unsigned long *pulData)
- void SSIDataPut (unsigned long ulBase, unsigned long ulData)
- long SSIDataPutNonBlocking (unsigned long ulBase, unsigned long ulData)
- void SSIDisable (unsigned long ulBase)
- void SSIDMADisable (unsigned long ulBase, unsigned long ulDMAFlags)
- void SSIDMAEnable (unsigned long ulBase, unsigned long ulDMAFlags)
- void SSIEnable (unsigned long ulBase)
- void SSIIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void SSIIntDisable (unsigned long ulBase, unsigned long ulIntFlags)

- void SSIIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- void SSIIntRegister (unsigned long ulBase, void (*pfnHandler)(void))
- unsigned long SSIIntStatus (unsigned long ulBase, tBoolean bMasked)
- void SSIIntUnregister (unsigned long ulBase)

18.2.1 Detailed Description

The SSI API is broken into 3 groups of functions: those that deal with configuration and state, those that handle data, and those that manage interrupts.

The configuration of the SSI module is managed by the SSIConfigSetExpClk() function, while state is managed by the SSIEnable() and SSIDisable() functions. The DMA interface is enabled or disabled by the SSIDMAEnable() and SSIDMADisable() functions.

Data handling is performed by the SSIDataPut(), SSIDataPutNonBlocking(), SSIDataGet(), and SSIDataGetNonBlocking() functions.

Interrupts from the SSI module are managed using the SSIIntClear(), SSIIntDisable(), SSIIntEnable(), SSIIntRegister(), SSIIntStatus(), and SSIIntUnregister() functions.

The SSIConfig(), SSIDataNonBlockingGet(), and SSIDataNonBlockingPut() APIs from previous versions of the peripheral driver library have been replaced by the SSIConfigSetExpClk(), SSIDataGetNonBlocking(), and SSIDataPutNonBlocking() APIs. Macros have been provided in `ssi.h` to map the old APIs to the new APIs, allowing existing applications to link and run with the new APIs. It is recommended that new applications utilize the new APIs in favor of the old ones.

18.2.2 Function Documentation

18.2.2.1 SSIConfigSetExpClk

Configures the synchronous serial interface.

Prototype:

```
void
SSIConfigSetExpClk(unsigned long ulBase,
                   unsigned long ulSSIClk,
                   unsigned long ulProtocol,
                   unsigned long ulMode,
                   unsigned long ulBitRate,
                   unsigned long ulDataWidth)
```

Parameters:

ulBase specifies the SSI module base address.

ulSSIClk is the rate of the clock supplied to the SSI module.

ulProtocol specifies the data transfer protocol.

ulMode specifies the mode of operation.

ulBitRate specifies the clock rate.

ulDataWidth specifies number of bits transfered per frame.

Description:

This function configures the synchronous serial interface. It sets the SSI protocol, mode of operation, bit rate, and data width.

The *ulProtocol* parameter defines the data frame format. The *ulProtocol* parameter can be one of the following values: **SSI_FRF_MOTO_MODE_0**, **SSI_FRF_MOTO_MODE_1**, **SSI_FRF_MOTO_MODE_2**, **SSI_FRF_MOTO_MODE_3**, **SSI_FRF_TI**, or **SSI_FRF_NMW**. The Motorola frame formats imply the following polarity and phase configurations:

```
Polarity Phase       Mode
  0       0    SSI_FRF_MOTO_MODE_0
  0       1    SSI_FRF_MOTO_MODE_1
  1       0    SSI_FRF_MOTO_MODE_2
  1       1    SSI_FRF_MOTO_MODE_3
```

The *ulMode* parameter defines the operating mode of the SSI module. The SSI module can operate as a master or slave; if a slave, the SSI can be configured to disable output on its serial output line. The *ulMode* parameter can be one of the following values: **SSI_MODE_MASTER**, **SSI_MODE_SLAVE**, or **SSI_MODE_SLAVE_OD**.

The *ulBitRate* parameter defines the bit rate for the SSI. This bit rate must satisfy the following clock ratio criteria:

- FSSI >= 2 * bit rate (master mode)
- FSSI >= 12 * bit rate (slave modes)

where FSSI is the frequency of the clock supplied to the SSI module.

The *ulDataWidth* parameter defines the width of the data transfers, and can be a value between 4 and 16, inclusive.

The peripheral clock will be the same as the processor clock. This will be the value returned by SysCtlClockGet(), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to SysCtlClockGet()).

This function replaces the original SSIConfig() API and performs the same actions. A macro is provided in `ssi.h` to map the original API to this API.

Returns:

None.

18.2.2.2 SSIDataGet

Gets a data element from the SSI receive FIFO.

Prototype:

```
void
SSIDataGet(unsigned long ulBase,
           unsigned long *pulData)
```

Parameters:

ulBase specifies the SSI module base address.

pulData pointer to a storage location for data that was received over the SSI interface.

Description:
This function will get received data from the receive FIFO of the specified SSI module, and place that data into the location specified by the *pulData* parameter.

Note:
Only the lower N bits of the value written to *pulData* will contain valid data, where N is the data width as configured by SSIConfigSetExpClk(). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *pulData* will contain valid data.

Returns:
None.

18.2.2.3 SSIDataGetNonBlocking

Gets a data element from the SSI receive FIFO.

Prototype:
```
long
SSIDataGetNonBlocking(unsigned long ulBase,
                      unsigned long *pulData)
```

Parameters:
ulBase specifies the SSI module base address.
pulData pointer to a storage location for data that was received over the SSI interface.

Description:
This function will get received data from the receive FIFO of the specified SSI module, and place that data into the location specified by the *ulData* parameter. If there is no data in the FIFO, then this function will return a zero.

This function replaces the original SSIDataNonBlockingGet() API and performs the same actions. A macro is provided in `ssi.h` to map the original API to this API.

Note:
Only the lower N bits of the value written to *pulData* will contain valid data, where N is the data width as configured by SSIConfigSetExpClk(). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *pulData* will contain valid data.

Returns:
Returns the number of elements read from the SSI receive FIFO.

18.2.2.4 SSIDataPut

Puts a data element into the SSI transmit FIFO.

Prototype:
```
void
SSIDataPut(unsigned long ulBase,
           unsigned long ulData)
```

Parameters:
ulBase specifies the SSI module base address.

ulData data to be transmitted over the SSI interface.

Description:
This function will place the supplied data into the transmit FIFO of the specified SSI module.

Note:
The upper 32 - N bits of the *ulData* will be discarded by the hardware, where N is the data width as configured by SSIConfigSetExpClk(). For example, if the interface is configured for 8-bit data width, the upper 24 bits of *ulData* will be discarded.

Returns:
None.

18.2.2.5 SSIDataPutNonBlocking

Puts a data element into the SSI transmit FIFO.

Prototype:
```
long
SSIDataPutNonBlocking(unsigned long ulBase,
                      unsigned long ulData)
```

Parameters:
ulBase specifies the SSI module base address.
ulData data to be transmitted over the SSI interface.

Description:
This function will place the supplied data into the transmit FIFO of the specified SSI module. If there is no space in the FIFO, then this function will return a zero.

This function replaces the original SSIDataNonBlockingPut() API and performs the same actions. A macro is provided in `ssi.h` to map the original API to this API.

Note:
The upper 32 - N bits of the *ulData* will be discarded by the hardware, where N is the data width as configured by SSIConfigSetExpClk(). For example, if the interface is configured for 8-bit data width, the upper 24 bits of *ulData* will be discarded.

Returns:
Returns the number of elements written to the SSI transmit FIFO.

18.2.2.6 SSIDisable

Disables the synchronous serial interface.

Prototype:
```
void
SSIDisable(unsigned long ulBase)
```

Parameters:
ulBase specifies the SSI module base address.

Description:
This will disable operation of the synchronous serial interface.

Returns:
None.

18.2.2.7 SSIDMADisable

Disable SSI DMA operation.

Prototype:
```
void
SSIDMADisable(unsigned long ulBase,
              unsigned long ulDMAFlags)
```

Parameters:
ulBase is the base address of the SSI port.
ulDMAFlags is a bit mask of the DMA features to disable.

Description:
This function is used to disable SSI DMA features that were enabled by SSIDMAEnable(). The specified SSI DMA features are disabled. The *ulDMAFlags* parameter is the logical OR of any of the following values:

- SSI_DMA_RX - disable DMA for receive
- SSI_DMA_TX - disable DMA for transmit

Returns:
None.

18.2.2.8 SSIDMAEnable

Enable SSI DMA operation.

Prototype:
```
void
SSIDMAEnable(unsigned long ulBase,
             unsigned long ulDMAFlags)
```

Parameters:
ulBase is the base address of the SSI port.
ulDMAFlags is a bit mask of the DMA features to enable.

Description:
The specified SSI DMA features are enabled. The SSI can be configured to use DMA for transmit and/or receive data transfers. The *ulDMAFlags* parameter is the logical OR of any of the following values:

- SSI_DMA_RX - enable DMA for receive
- SSI_DMA_TX - enable DMA for transmit

Note:
The uDMA controller must also be set up before DMA can be used with the SSI.

Returns:
None.

18.2.2.9 SSIEnable

Enables the synchronous serial interface.

Prototype:
```
void
SSIEnable(unsigned long ulBase)
```

Parameters:
ulBase specifies the SSI module base address.

Description:
This will enable operation of the synchronous serial interface. It must be configured before it is enabled.

Returns:
None.

18.2.2.10 SSIIntClear

Clears SSI interrupt sources.

Prototype:
```
void
SSIIntClear(unsigned long ulBase,
            unsigned long ulIntFlags)
```

Parameters:
ulBase specifies the SSI module base address.
ulIntFlags is a bit mask of the interrupt sources to be cleared.

Description:
The specified SSI interrupt sources are cleared, so that they no longer assert. This must be done in the interrupt handler to keep it from being called again immediately upon exit. The *ulIntFlags* parameter can consist of either or both the **SSI_RXTO** and **SSI_RXOR** values.

Note:
Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:
None.

18.2.2.11 SSIIntDisable

Disables individual SSI interrupt sources.

Prototype:

```
void
SSIIntDisable(unsigned long ulBase,
              unsigned long ulIntFlags)
```

Parameters:

ulBase specifies the SSI module base address.

ulIntFlags is a bit mask of the interrupt sources to be disabled.

Description:

Disables the indicated SSI interrupt sources. The *ulIntFlags* parameter can be any of the **SSI_TXFF**, **SSI_RXFF**, **SSI_RXTO**, or **SSI_RXOR** values.

Returns:

None.

18.2.2.12 SSIIntEnable

Enables individual SSI interrupt sources.

Prototype:

```
void
SSIIntEnable(unsigned long ulBase,
             unsigned long ulIntFlags)
```

Parameters:

ulBase specifies the SSI module base address.

ulIntFlags is a bit mask of the interrupt sources to be enabled.

Description:

Enables the indicated SSI interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. The *ulIntFlags* parameter can be any of the **SSI_TXFF**, **SSI_RXFF**, **SSI_RXTO**, or **SSI_RXOR** values.

Returns:

None.

18.2.2.13 SSIIntRegister

Registers an interrupt handler for the synchronous serial interface.

Prototype:

```
void
SSIIntRegister(unsigned long ulBase,
               void (*pfnHandler)(void))
```

Parameters:

ulBase specifies the SSI module base address.

pfnHandler is a pointer to the function to be called when the synchronous serial interface interrupt occurs.

Description:

This sets the handler to be called when an SSI interrupt occurs. This will enable the global interrupt in the interrupt controller; specific SSI interrupts must be enabled via SSIIntEnable(). If necessary, it is the interrupt handler's responsibility to clear the interrupt source via SSIIntClear().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

18.2.2.14 SSIIntStatus

Gets the current interrupt status.

Prototype:

```
unsigned long
SSIIntStatus(unsigned long ulBase,
             tBoolean bMasked)
```

Parameters:

ulBase specifies the SSI module base address.

bMasked is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

Description:

This returns the interrupt status for the SSI module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

The current interrupt status, enumerated as a bit field of **SSI_TXFF**, **SSI_RXFF**, **SSI_RXTO**, and **SSI_RXOR**.

18.2.2.15 SSIIntUnregister

Unregisters an interrupt handler for the synchronous serial interface.

Prototype:

```
void
SSIIntUnregister(unsigned long ulBase)
```

Parameters:

ulBase specifies the SSI module base address.

Description:

This function will clear the handler to be called when a SSI interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

18.3 Programming Example

The following example shows how to use the SSI API to configure the SSI module as a master device, and how to do a simple send of data.

```
char *pcChars = "SSI Master send data.";
long lIdx;

//
// Configure the SSI.
//
SSIConfigSetExpClk(SSI_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE0,
                   SSI_MODE_MASTER, 2000000, 8);

//
// Enable the SSI module.
//
SSIEnable(SSI_BASE);

//
// Send some data.
//
lIdx = 0;
while(pcChars[lIdx])
{
    if(SSIDataPut(SSI_BASE, pcChars[lIdx]))
    {
        lIdx++;
    }
}
```

19 System Control

19.1 Introduction

System control determines the overall operation of the device. It controls the clocking of the device, the set of peripherals that are enabled, configuration of the device and its resets, and provides information about the device.

The members of the Stellaris family have a varying peripheral set and memory sizes. The device has a set of read-only registers that indicate the size of the memories, the peripherals that are present, and the pins that are present for peripherals that have a varying number of pins. This information can be used to write adaptive software that will run on more than one member of the Stellaris family.

The device can be clocked from one of five sources: an external oscillator, the main oscillator, the internal oscillator, the internal oscillator divided by four, or the PLL. The PLL can use any of the four oscillators as its input. Since the internal oscillator has a very wide error range (+/- 50%), it cannot be used for applications that require specific timing; its real use is for detecting failures of the main oscillator and the PLL, and for applications that strictly respond to external events and do not use time-based peripherals (such as a UART). When using the PLL, the input clock frequency is constrained to specific frequencies between 3.579545 MHz and 8.192 MHz (that is, the standard crystal frequencies in that range). When direct clocking with an external oscillator or the main oscillator, the frequency is constrained to between 0 Hz and 50 MHz (depending on the part). The internal oscillator is 15 MHz, +/- 50%; its frequency will vary by device, with voltage, and with temperature. The internal oscillator provides no tuning or frequency measurement mechanism; its frequency is not adjustable.

Almost the entire device operates from a single clock. The ADC and PWM blocks have their own clocks. In order to use the ADC, the PLL must be used; the PLL output will be used to create the clock required by the ADC. The PWM has its own optional divider from the system clock; this can be power of two divides between 1 and 64.

Three modes of operation are supported by the Stellaris family: run mode, sleep mode, and deep-sleep mode. In run mode, the processor is actively executing code. In sleep mode, the clocking of the device is unchanged but the processor no longer executes code (and is no longer clocked). In deep-sleep mode, the clocking of the device may change (depending upon the run mode clock configuration) and the processor no longer executes code (and is no longer clocked). An interrupt will return the device to run mode from one of the sleep modes; the sleep modes are entered upon request from the code.

The device has an internal LDO for generating the on-chip 2.5 V power supply; the output voltage of the LDO can be adjusted between 2.25 V and 2.75 V. Depending upon the application, lower voltage may be advantageous for its power savings, or higher voltage may be advantageous for its improved performance. The default setting of 2.5 V is a good compromise between the two, and should not be changed without careful consideration and evaluation.

There are several system events that, when detected, will cause system control to reset the device. These events are the input voltage dropping too low, the LDO voltage dropping too low, an external

reset, a software reset request, and a watchdog timeout. The properties of some of these events can be configured, and the reason for a reset can be determined from system control.

Each peripheral in the device can be individually enabled, disabled, or reset. Additionally, the set of peripherals that remain enabled during sleep mode and deep-sleep mode can be configured, allowing custom sleep and deep-sleep modes to be defined. Care must be taken with deep-sleep mode, though, since in this mode the PLL is no longer used and the system is clocked by the input crystal. Peripherals that depend upon a particular input clock rate (such as a UART) will not operate as expected in deep-sleep mode due to the clock rate change; these peripherals must either be reconfigured upon entry to and exit from deep-sleep mode, or simply not enabled in deep-sleep mode.

There are various system events that, when detected, will cause system control to generate a processor interrupt. These events are the PLL achieving lock, the internal LDO current limit being exceeded, the internal oscillator failing, the main oscillator failing, the input voltage dropping too low, the internal LDO voltage dropping too low, and the PLL failing. Each of these interrupts can be individually enabled or disabled, and the sources must be cleared by the interrupt handler when they occur.

This driver is contained in `src/sysctl.c`, with `src/sysctl.h` containing the API definitions for use by applications.

19.2 API Functions

Functions

- unsigned long SysCtlADCSpeedGet (void)
- void SysCtlADCSpeedSet (unsigned long ulSpeed)
- void SysCtlBrownOutConfigSet (unsigned long ulConfig, unsigned long ulDelay)
- void SysCtlClkVerificationClear (void)
- unsigned long SysCtlClockGet (void)
- void SysCtlClockSet (unsigned long ulConfig)
- void SysCtlDeepSleep (void)
- void SysCtlDelay (unsigned long ulCount)
- unsigned long SysCtlFlashSizeGet (void)
- void SysCtlGPIOAHBDisable (unsigned long ulGPIOPeripheral)
- void SysCtlGPIOAHBEnable (unsigned long ulGPIOPeripheral)
- void SysCtlIntClear (unsigned long ulInts)
- void SysCtlIntDisable (unsigned long ulInts)
- void SysCtlIntEnable (unsigned long ulInts)
- void SysCtlIntRegister (void (*pfnHandler)(void))
- unsigned long SysCtlIntStatus (tBoolean bMasked)
- void SysCtlIntUnregister (void)
- void SysCtlIOSCVerificationSet (tBoolean bEnable)
- void SysCtlLDOConfigSet (unsigned long ulConfig)
- unsigned long SysCtlLDOGet (void)
- void SysCtlLDOSet (unsigned long ulVoltage)
- void SysCtlMOSCVerificationSet (tBoolean bEnable)

- void SysCtlPeripheralClockGating (tBoolean bEnable)
- void SysCtlPeripheralDeepSleepDisable (unsigned long ulPeripheral)
- void SysCtlPeripheralDeepSleepEnable (unsigned long ulPeripheral)
- void SysCtlPeripheralDisable (unsigned long ulPeripheral)
- void SysCtlPeripheralEnable (unsigned long ulPeripheral)
- tBoolean SysCtlPeripheralPresent (unsigned long ulPeripheral)
- void SysCtlPeripheralReset (unsigned long ulPeripheral)
- void SysCtlPeripheralSleepDisable (unsigned long ulPeripheral)
- void SysCtlPeripheralSleepEnable (unsigned long ulPeripheral)
- tBoolean SysCtlPinPresent (unsigned long ulPin)
- void SysCtlPLLVerificationSet (tBoolean bEnable)
- unsigned long SysCtlPWMClockGet (void)
- void SysCtlPWMClockSet (unsigned long ulConfig)
- void SysCtlReset (void)
- void SysCtlResetCauseClear (unsigned long ulCauses)
- unsigned long SysCtlResetCauseGet (void)
- void SysCtlSleep (void)
- unsigned long SysCtlSRAMSizeGet (void)
- void SysCtlUSBPLLDisable (void)
- void SysCtlUSBPLLEnable (void)

19.2.1 Detailed Description

The SysCtl API is broken up into eight groups of functions: those that provide device information, those that deal with device clocking, those that provide peripheral control, those that deal with the SysCtl interrupt, those that deal with the LDO, those that deal with sleep modes, those that deal with reset reasons, those that deal with the brown-out reset, and those that deal with clock verification timers.

Information about the device is provided by SysCtlSRAMSizeGet(), SysCtlFlashSizeGet(), SysCtlPeripheralPresent(), and SysCtlPinPresent().

Clocking of the device is configured with SysCtlClockSet() and SysCtlPWMClockSet(). Information about device clocking is provided by SysCtlClockGet() and SysCtlPWMClockGet().

Peripheral enabling and reset are controlled with SysCtlPeripheralReset(), SysCtlPeripheralEnable(), SysCtlPeripheralDisable(), SysCtlPeripheralSleepEnable(), SysCtlPeripheralSleepDisable(), SysCtlPeripheralDeepSleepEnable(), SysCtlPeripheralDeepSleepDisable(), and SysCtlPeripheralClockGating().

The system control interrupt is managed with SysCtlIntRegister(), SysCtlIntUnregister(), SysCtlIntEnable(), SysCtlIntDisable(), SysCtlIntClear(), SysCtlIntStatus().

The LDO is controlled with SysCtlLDOSet() and SysCtlLDOConfigSet(). Its status is provided by SysCtlLDOGet().

The device is put into sleep modes with SysCtlSleep() and SysCtlDeepSleep().

The reset reason is managed with SysCtlResetCauseGet() and SysCtlResetCauseClear(). A software reset is performed with SysCtlReset().

The brown-out reset is configured with SysCtlBrownOutConfigSet().

The clock verification timers are managed with SysCtlIOSCVerificationSet(), SysCtlMOSCVerificationSet(), SysCtlPLLVerificationSet(), and SysCtlClkVerificationClear().

19.2.2 Function Documentation

19.2.2.1 SysCtlADCSpeedGet

Gets the sample rate of the ADC.

Prototype:

```
unsigned long
SysCtlADCSpeedGet(void)
```

Description:

This function gets the current sample rate of the ADC.

Returns:

Returns the current ADC sample rate; will be one of **SYSCTL_ADCSPEED_1MSPS**, **SYSCTL_ADCSPEED_500KSPS**, **SYSCTL_ADCSPEED_250KSPS**, or **SYSCTL_ADCSPEED_125KSPS**.

19.2.2.2 SysCtlADCSpeedSet

Sets the sample rate of the ADC.

Prototype:

```
void
SysCtlADCSpeedSet(unsigned long ulSpeed)
```

Parameters:

ulSpeed is the desired sample rate of the ADC; must be one of **SYSCTL_ADCSPEED_1MSPS**, **SYSCTL_ADCSPEED_500KSPS**, **SYSCTL_ADCSPEED_250KSPS**, or **SYSCTL_ADCSPEED_125KSPS**.

Description:

This function sets the rate at which the ADC samples are captured by the ADC block. The sampling speed may be limited by the hardware, so the sample rate may end up being slower than requested. SysCtlADCSpeedGet() will return the actual speed in use.

Returns:

None.

19.2.2.3 SysCtlBrownOutConfigSet

Configures the brown-out control.

Prototype:

```
void
SysCtlBrownOutConfigSet(unsigned long ulConfig,
                        unsigned long ulDelay)
```

Parameters:

ulConfig is the desired configuration of the brown-out control. Must be the logical OR of **SYSCTL_BOR_RESET** and/or **SYSCTL_BOR_RESAMPLE**.

ulDelay is the number of internal oscillator cycles to wait before resampling an asserted brown-out signal. This value only has meaning when **SYSCTL_BOR_RESAMPLE** is set and must be less than 8192.

Description:

This function configures how the brown-out control operates. It can detect a brown-out by looking at only the brown-out output, or it can wait for it to be active for two consecutive samples separated by a configurable time. When it detects a brown-out condition, it can either reset the device or generate a processor interrupt.

Returns:

None.

19.2.2.4 SysCtlClkVerificationClear

Clears the clock verification status.

Prototype:

```
void
SysCtlClkVerificationClear(void)
```

Description:

This function clears the status of the clock verification timers, allowing them to assert another failure if detected.

The clock verification timers are only available on Sandstorm-class devices.

Returns:

None.

19.2.2.5 SysCtlClockGet

Gets the processor clock rate.

Prototype:

```
unsigned long
SysCtlClockGet(void)
```

Description:

This function determines the clock rate of the processor clock. This is also the clock rate of all the peripheral modules (with the exception of PWM, which has its own clock divider).

Note:

This will not return accurate results if SysCtlClockSet() has not been called to configure the clocking of the device, or if the device is directly clocked from a crystal (or a clock source) that is not one of the supported crystal frequencies. In the later case, this function should be modified to directly return the correct system clock rate.

Returns:

The processor clock rate.

19.2.2.6 SysCtlClockSet

Sets the clocking of the device.

Prototype:

```
void
SysCtlClockSet(unsigned long ulConfig)
```

Parameters:

ulConfig is the required configuration of the device clocking.

Description:

This function configures the clocking of the device. The input crystal frequency, oscillator to be used, use of the PLL, and the system clock divider are all configured with this function.

The *ulConfig* parameter is the logical OR of several different values, many of which are grouped into sets where only one can be chosen.

The system clock divider is chosen with one of the following values: **SYSCTL_SYSDIV_1**, **SYSCTL_SYSDIV_2**, **SYSCTL_SYSDIV_3**, ... **SYSCTL_SYSDIV_64**. Only **SYSCTL_SYSDIV_1** through **SYSCTL_SYSDIV_16** are valid on Sandstorm-class devices.

The use of the PLL is chosen with either **SYSCTL_USE_PLL** or **SYSCTL_USE_OSC**.

The external crystal frequency is chosen with one of the following values: **SYSCTL_XTAL_1MHZ**, **SYSCTL_XTAL_1_84MHZ**, **SYSCTL_XTAL_2MHZ**, **SYSCTL_XTAL_2_45MHZ**, **SYSCTL_XTAL_3_57MHZ**, **SYSCTL_XTAL_3_68MHZ**, **SYSCTL_XTAL_4MHZ**, **SYSCTL_XTAL_4_09MHZ**, **SYSCTL_XTAL_4_91MHZ**, **SYSCTL_XTAL_5MHZ**, **SYSCTL_XTAL_5_12MHZ**, **SYSCTL_XTAL_6MHZ**, **SYSCTL_XTAL_6_14MHZ**, **SYSCTL_XTAL_7_37MHZ**, **SYSCTL_XTAL_8MHZ**, **SYSCTL_XTAL_8_19MHZ**, **SYSCTL_XTAL_10MHZ**, **SYSCTL_XTAL_12MHZ**, **SYSCTL_XTAL_12_2MHZ**, **SYSCTL_XTAL_13_5MHZ**, **SYSCTL_XTAL_14_3MHZ**, **SYSCTL_XTAL_16MHZ**, or **SYSCTL_XTAL_16_3MHZ**. Values below **SYSCTL_XTAL_3_57MHZ** are not valid when the PLL is in operation. On Sandstorm- and Fury-class devices, values above **SYSCTL_XTAL_8_19MHZ** are not valid.

The oscillator source is chosen with one of the following values: **SYSCTL_OSC_MAIN**, **SYSCTL_OSC_INT**, **SYSCTL_OSC_INT4**, **SYSCTL_OSC_INT30**, or **SYSCTL_OSC_EXT32**. On Sandstorm-class devices, **SYSCTL_OSC_INT30** and **SYSCTL_OSC_EXT32** are not valid. **SYSCTL_OSC_EXT32** is only available on devices with the hibernate module, and then only when the hibernate module has been enabled.

The internal and main oscillators are disabled with the **SYSCTL_INT_OSC_DIS** and **SYSCTL_MAIN_OSC_DIS** flags, respectively. The external oscillator must be enabled in order to use an external clock source. Note that attempts to disable the oscillator used to clock the device will be prevented by the hardware.

To clock the system from an external source (such as an external crystal oscillator), use **SYSCTL_USE_OSC** | **SYSCTL_OSC_MAIN**. To clock the system from the main oscillator, use **SYSCTL_USE_OSC** | **SYSCTL_OSC_MAIN**. To clock the system from the PLL, use **SYSCTL_USE_PLL** | **SYSCTL_OSC_MAIN**, and select the appropriate crystal with one of the **SYSCTL_XTAL_xxx** values.

Note:

If selecting the PLL as the system clock source (that is, via **SYSCTL_USE_PLL**), this function will poll the PLL lock interrupt to determine when the PLL has locked. If an interrupt handler for the system control interrupt is in place, and it responds to and clears the PLL lock interrupt,

this function will delay until its timeout has occurred instead of completing as soon as PLL lock is achieved.

Returns:
None.

19.2.2.7 SysCtlDeepSleep

Puts the processor into deep-sleep mode.

Prototype:
```
void
SysCtlDeepSleep(void)
```

Description:
This function places the processor into deep-sleep mode; it will not return until the processor returns to run mode. The peripherals that are enabled via SysCtlPeripheralDeepSleepEnable() continue to operate and can wake up the processor (if automatic clock gating is enabled with SysCtlPeripheralClockGating(), otherwise all peripherals continue to operate).

Returns:
None.

19.2.2.8 SysCtlDelay

Provides a small delay.

Prototype:
```
void
SysCtlDelay(unsigned long ulCount)
```

Parameters:
ulCount is the number of delay loop iterations to perform.

Description:
This function provides a means of generating a constant length delay. It is written in assembly to keep the delay consistent across tool chains, avoiding the need to tune the delay based on the tool chain in use.

The loop takes 3 cycles/loop.

Returns:
None.

19.2.2.9 SysCtlFlashSizeGet

Gets the size of the flash.

Prototype:
```
unsigned long
SysCtlFlashSizeGet(void)
```

Description:
This function determines the size of the flash on the Stellaris device.

Returns:
The total number of bytes of flash.

19.2.2.10 SysCtlGPIOAHBDisable

Disables a GPIO peripheral for access from the high speed bus.

Prototype:
```
void
SysCtlGPIOAHBDisable(unsigned long ulGPIOPeripheral)
```

Parameters:
ulGPIOPeripheral is the GPIO peripheral to disable.

Description:
This function will disable the specified GPIO peripherals for access from the high speed bus. Once disabled, the GPIO peripheral is accessed from the peripheral bus.

The **ulGPIOPeripheral** argument must be only one of the following values: **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_GPIOF**, **SYSCTL_PERIPH_GPIOG**, or **SYSCTL_PERIPH_GPIOH**.

Returns:
None.

19.2.2.11 SysCtlGPIOAHBEnable

Enables a GPIO peripheral for access from the high speed bus.

Prototype:
```
void
SysCtlGPIOAHBEnable(unsigned long ulGPIOPeripheral)
```

Parameters:
ulGPIOPeripheral is the GPIO peripheral to enable.

Description:
This function is used to enable the specified GPIO peripherals to be accessed from the high speed bus instead of the peripheral bus. When a GPIO peripheral is enabled for high speed access, the **_AHB_BASE** form of the base address should be used for GPIO functions. For example, instead of using **GPIO_PORTA_BASE** as the base address for GPIO functions, use **GPIO_PORTA_AHB_BASE** instead.

The *ulGPIOPeripheral* argument must be only one of the following values: **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_GPIOF**, **SYSCTL_PERIPH_GPIOG**, or **SYSCTL_PERIPH_GPIOH**.

Returns:
None.

19.2.2.12 SysCtlIntClear

Clears system control interrupt sources.

Prototype:
```
void
SysCtlIntClear(unsigned long ulInts)
```

Parameters:
ulInts is a bit mask of the interrupt sources to be cleared. Must be a logical OR of **SYSCTL_INT_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_PLL_FAIL**.

Description:
The specified system control interrupt sources are cleared, so that they no longer assert. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

Note:
Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:
None.

19.2.2.13 SysCtlIntDisable

Disables individual system control interrupt sources.

Prototype:
```
void
SysCtlIntDisable(unsigned long ulInts)
```

Parameters:
ulInts is a bit mask of the interrupt sources to be disabled. Must be a logical OR of **SYSCTL_INT_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_PLL_FAIL**.

Description:
Disables the indicated system control interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns:
None.

19.2.2.14 SysCtlIntEnable

Enables individual system control interrupt sources.

Prototype:

```
void
SysCtlIntEnable(unsigned long ulInts)
```

Parameters:

ulInts is a bit mask of the interrupt sources to be enabled. Must be a logical OR of **SYSCTL_INT_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and/or **SYSCTL_INT_PLL_FAIL**.

Description:

Enables the indicated system control interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

Returns:

None.

19.2.2.15 SysCtlIntRegister

Registers an interrupt handler for the system control interrupt.

Prototype:

```
void
SysCtlIntRegister(void (*pfnHandler)(void))
```

Parameters:

pfnHandler is a pointer to the function to be called when the system control interrupt occurs.

Description:

This sets the handler to be called when a system control interrupt occurs. This will enable the global interrupt in the interrupt controller; specific system control interrupts must be enabled via SysCtlIntEnable(). It is the interrupt handler's responsibility to clear the interrupt source via SysCtlIntClear().

System control can generate interrupts when the PLL achieves lock, if the internal LDO current limit is exceeded, if the internal oscillator fails, if the main oscillator fails, if the internal LDO output voltage droops too much, if the external voltage droops too much, or if the PLL fails.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

19.2.2.16 SysCtlIntStatus

Gets the current interrupt status.

Prototype:

```
unsigned long
SysCtlIntStatus(tBoolean bMasked)
```

Parameters:

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This returns the interrupt status for the system controller. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

The current interrupt status, enumerated as a bit field of **SYSCTL_INT_PLL_LOCK**, **SYSCTL_INT_CUR_LIMIT**, **SYSCTL_INT_IOSC_FAIL**, **SYSCTL_INT_MOSC_FAIL**, **SYSCTL_INT_POR**, **SYSCTL_INT_BOR**, and **SYSCTL_INT_PLL_FAIL**.

19.2.2.17 SysCtlIntUnregister

Unregisters the interrupt handler for the system control interrupt.

Prototype:

```
void
SysCtlIntUnregister(void)
```

Description:

This function will clear the handler to be called when a system control interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

19.2.2.18 SysCtlIOSCVerificationSet

Configures the internal oscillator verification timer.

Prototype:

```
void
SysCtlIOSCVerificationSet(tBoolean bEnable)
```

Parameters:

bEnable is a boolean that is **true** if the internal oscillator verification timer should be enabled.

Description:

This function allows the internal oscillator verification timer to be enabled or disabled. When enabled, an interrupt will be generated if the internal oscillator ceases to operate.

The internal oscillator verification timer is only available on Sandstorm-class devices.

Note:
Both oscillators (main and internal) must be enabled for this verification timer to operate as the main oscillator will verify the internal oscillator.

Returns:
None.

19.2.2.19 SysCtlLDOConfigSet

Configures the LDO failure control.

Prototype:
```
void
SysCtlLDOConfigSet(unsigned long ulConfig)
```

Parameters:
ulConfig is the required LDO failure control setting; can be either **SYSCTL_LDOCFG_ARST** or **SYSCTL_LDOCFG_NORST**.

Description:
This function allows the LDO to be configured to cause a processor reset when the output voltage becomes unregulated.

The LDO failure control is only available on Sandstorm-class devices.

Returns:
None.

19.2.2.20 SysCtlLDOGet

Gets the output voltage of the LDO.

Prototype:
```
unsigned long
SysCtlLDOGet(void)
```

Description:
This function determines the output voltage of the LDO, as specified by the control register.

Returns:
Returns the current voltage of the LDO; will be one of **SYSCTL_LDO_2_25V**, **SYSCTL_LDO_2_30V**, **SYSCTL_LDO_2_35V**, **SYSCTL_LDO_2_40V**, **SYSCTL_LDO_2_45V**, **SYSCTL_LDO_2_50V**, **SYSCTL_LDO_2_55V**, **SYSCTL_LDO_2_60V**, **SYSCTL_LDO_2_65V**, **SYSCTL_LDO_2_70V**, or **SYSCTL_LDO_2_75V**.

19.2.2.21 SysCtlLDOSet

Sets the output voltage of the LDO.

Prototype:

```
void
SysCtlLDOSet(unsigned long ulVoltage)
```

Parameters:

ulVoltage is the required output voltage from the LDO. Must be one of **SYSCTL_LDO_2_25V**, **SYSCTL_LDO_2_30V**, **SYSCTL_LDO_2_35V**, **SYSCTL_LDO_2_40V**, **SYSCTL_LDO_2_45V**, **SYSCTL_LDO_2_50V**, **SYSCTL_LDO_2_55V**, **SYSCTL_LDO_2_60V**, **SYSCTL_LDO_2_65V**, **SYSCTL_LDO_2_70V**, or **SYSCTL_LDO_2_75V**.

Description:

This function sets the output voltage of the LDO. The default voltage is 2.5 V; it can be adjusted +/- 10%.

Returns:

None.

19.2.2.22 SysCtlMOSCVerificationSet

Configures the main oscillator verification timer.

Prototype:

```
void
SysCtlMOSCVerificationSet(tBoolean bEnable)
```

Parameters:

bEnable is a boolean that is **true** if the main oscillator verification timer should be enabled.

Description:

This function allows the main oscillator verification timer to be enabled or disabled. When enabled, an interrupt will be generated if the main oscillator ceases to operate.

The main oscillator verification timer is only available on Sandstorm-class devices.

Note:

Both oscillators (main and internal) must be enabled for this verification timer to operate as the internal oscillator will verify the main oscillator.

Returns:

None.

19.2.2.23 SysCtlPeripheralClockGating

Controls peripheral clock gating in sleep and deep-sleep mode.

Prototype:

```
void
SysCtlPeripheralClockGating(tBoolean bEnable)
```

Parameters:

bEnable is a boolean that is **true** if the sleep and deep-sleep peripheral configuration should be used and **false** if not.

Description:

This function controls how peripherals are clocked when the processor goes into sleep or deep-sleep mode. By default, the peripherals are clocked the same as in run mode; if peripheral clock gating is enabled they are clocked according to the configuration set by SysCtlPeripheralSleepEnable(), SysCtlPeripheralSleepDisable(), SysCtlPeripheralDeepSleepEnable(), and SysCtlPeripheralDeepSleepDisable().

Returns:

None.

19.2.2.24 SysCtlPeripheralDeepSleepDisable

Disables a peripheral in deep-sleep mode.

Prototype:

```
void
SysCtlPeripheralDeepSleepDisable(unsigned long ulPeripheral)
```

Parameters:

ulPeripheral is the peripheral to disable in deep-sleep mode.

Description:

This function causes a peripheral to stop operating when the processor goes into deep-sleep mode. Disabling peripherals while in deep-sleep mode helps to lower the current draw of the device, and can keep peripherals that require a particular clock frequency from operating when the clock changes as a result of entering deep-sleep mode. If enabled (via SysCtlPeripheralEnable()), the peripheral will automatically resume operation when the processor leaves deep-sleep mode, maintaining its entire state from before deep-sleep mode was entered.

Deep-sleep mode clocking of peripherals must be enabled via SysCtlPeripheralClockGating(); if disabled, the peripheral deep-sleep mode configuration is maintained but has no effect when deep-sleep mode is entered.

The *ulPeripheral* parameter must be one of the following values: **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_ETH**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_GPIOF**, **SYSCTL_PERIPH_GPIOG**, **SYSCTL_PERIPH_GPIOH**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_PWM**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, or **SYSCTL_PERIPH_WDOG**.

Returns:

None.

19.2.2.25 SysCtlPeripheralDeepSleepEnable

Enables a peripheral in deep-sleep mode.

Prototype:

```
void
SysCtlPeripheralDeepSleepEnable(unsigned long ulPeripheral)
```

Parameters:

ulPeripheral is the peripheral to enable in deep-sleep mode.

Description:

This function allows a peripheral to continue operating when the processor goes into deep-sleep mode. Since the clocking configuration of the device may change, not all peripherals can safely continue operating while the processor is in sleep mode. Those that must run at a particular frequency (such as a UART) will not work as expected if the clock changes. It is the responsibility of the caller to make sensible choices.

Deep-sleep mode clocking of peripherals must be enabled via SysCtlPeripheralClockGating(); if disabled, the peripheral deep-sleep mode configuration is maintained but has no effect when deep-sleep mode is entered.

The *ulPeripheral* parameter must be one of the following values: **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_ETH**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_GPIOF**, **SYSCTL_PERIPH_GPIOG**, **SYSCTL_PERIPH_GPIOH**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_PWM**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, or **SYSCTL_PERIPH_WDOG**.

Returns:

None.

19.2.2.26 SysCtlPeripheralDisable

Disables a peripheral.

Prototype:

```
void
SysCtlPeripheralDisable(unsigned long ulPeripheral)
```

Parameters:

ulPeripheral is the peripheral to disable.

Description:

Peripherals are disabled with this function. Once disabled, they will not operate or respond to register reads/writes.

The *ulPeripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_ETH**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**,

SYSCTL_PERIPH_GPIOF, **SYSCTL_PERIPH_GPIOG**, **SYSCTL_PERIPH_GPIOH**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_PWM**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, or **SYSCTL_PERIPH_WDOG**.

Returns:
None.

19.2.2.27 SysCtlPeripheralEnable

Enables a peripheral.

Prototype:
```
void
SysCtlPeripheralEnable(unsigned long ulPeripheral)
```

Parameters:
ulPeripheral is the peripheral to enable.

Description:
Peripherals are enabled with this function. At power-up, all peripherals are disabled; they must be enabled in order to operate or respond to register reads/writes.

The *ulPeripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_ETH**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_GPIOF**, **SYSCTL_PERIPH_GPIOG**, **SYSCTL_PERIPH_GPIOH**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_PWM**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, or **SYSCTL_PERIPH_WDOG**.

Note:
It takes five clock cycles after the write to enable a peripheral before the the peripheral is actually enabled. During this time, attempts to access the peripheral will result in a bus fault. Care should be taken to ensure that the peripheral is not accessed during this brief time period.

Returns:
None.

19.2.2.28 SysCtlPeripheralPresent

Determines if a peripheral is present.

Prototype:

```
tBoolean
SysCtlPeripheralPresent(unsigned long ulPeripheral)
```

Parameters:

ulPeripheral is the peripheral in question.

Description:

Determines if a particular peripheral is present in the device. Each member of the Stellaris family has a different peripheral set; this will determine which are present on this device.

The *ulPeripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_ETH**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_GPIOF**, **SYSCTL_PERIPH_GPIOG**, **SYSCTL_PERIPH_GPIOH**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_IEEE1588**, **SYSCTL_PERIPH_MPU**, **SYSCTL_PERIPH_PLL**, **SYSCTL_PERIPH_PWM**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TEMP**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, or **SYSCTL_PERIPH_WDOG**.

Returns:

Returns **true** if the specified peripheral is present and **false** if it is not.

19.2.2.29 SysCtlPeripheralReset

Performs a software reset of a peripheral.

Prototype:

```
void
SysCtlPeripheralReset(unsigned long ulPeripheral)
```

Parameters:

ulPeripheral is the peripheral to reset.

Description:

This function performs a software reset of the specified peripheral. An individual peripheral reset signal is asserted for a brief period and then deasserted, leaving the peripheral in a operating state but in its reset condition.

The *ulPeripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_ETH**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_GPIOF**, **SYSCTL_PERIPH_GPIOG**, **SYSCTL_PERIPH_GPIOH**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_PWM**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**,

SYSCTL_PERIPH_SSI0, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, or **SYSCTL_PERIPH_WDOG**.

Returns:
None.

19.2.2.30 SysCtlPeripheralSleepDisable

Disables a peripheral in sleep mode.

Prototype:
```
void
SysCtlPeripheralSleepDisable(unsigned long ulPeripheral)
```

Parameters:
ulPeripheral is the peripheral to disable in sleep mode.

Description:
This function causes a peripheral to stop operating when the processor goes into sleep mode. Disabling peripherals while in sleep mode helps to lower the current draw of the device. If enabled (via SysCtlPeripheralEnable()), the peripheral will automatically resume operation when the processor leaves sleep mode, maintaining its entire state from before sleep mode was entered.

Sleep mode clocking of peripherals must be enabled via SysCtlPeripheralClockGating(); if disabled, the peripheral sleep mode configuration is maintained but has no effect when sleep mode is entered.

The *ulPeripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_ETH**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_GPIOF**, **SYSCTL_PERIPH_GPIOG**, **SYSCTL_PERIPH_GPIOH**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_PWM**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, or **SYSCTL_PERIPH_WDOG**.

Returns:
None.

19.2.2.31 SysCtlPeripheralSleepEnable

Enables a peripheral in sleep mode.

Prototype:

```
void
SysCtlPeripheralSleepEnable(unsigned long ulPeripheral)
```

Parameters:

ulPeripheral is the peripheral to enable in sleep mode.

Description:

This function allows a peripheral to continue operating when the processor goes into sleep mode. Since the clocking configuration of the device does not change, any peripheral can safely continue operating while the processor is in sleep mode, and can therefore wake the processor from sleep mode.

Sleep mode clocking of peripherals must be enabled via SysCtlPeripheralClockGating(); if disabled, the peripheral sleep mode configuration is maintained but has no effect when sleep mode is entered.

The *ulPeripheral* parameter must be only one of the following values: **SYSCTL_PERIPH_ADC**, **SYSCTL_PERIPH_CAN0**, **SYSCTL_PERIPH_CAN1**, **SYSCTL_PERIPH_CAN2**, **SYSCTL_PERIPH_COMP0**, **SYSCTL_PERIPH_COMP1**, **SYSCTL_PERIPH_COMP2**, **SYSCTL_PERIPH_ETH**, **SYSCTL_PERIPH_GPIOA**, **SYSCTL_PERIPH_GPIOB**, **SYSCTL_PERIPH_GPIOC**, **SYSCTL_PERIPH_GPIOD**, **SYSCTL_PERIPH_GPIOE**, **SYSCTL_PERIPH_GPIOF**, **SYSCTL_PERIPH_GPIOG**, **SYSCTL_PERIPH_GPIOH**, **SYSCTL_PERIPH_HIBERNATE**, **SYSCTL_PERIPH_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_PWM**, **SYSCTL_PERIPH_QEI0**, **SYSCTL_PERIPH_QEI1**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, or **SYSCTL_PERIPH_WDOG**.

Returns:

None.

19.2.2.32 SysCtlPinPresent

Determines if a pin is present.

Prototype:

```
tBoolean
SysCtlPinPresent(unsigned long ulPin)
```

Parameters:

ulPin is the pin in question.

Description:

Determines if a particular pin is present in the device. The PWM, analog comparators, ADC, and timers have a varying number of pins across members of the Stellaris family; this will determine which are present on this device.

The *ulPin* argument must be only one of the following values: **SYSCTL_PIN_PWM0**, **SYSCTL_PIN_PWM1**, **SYSCTL_PIN_PWM2**, **SYSCTL_PIN_PWM3**, **SYSCTL_PIN_PWM4**, **SYSCTL_PIN_PWM5**, **SYSCTL_PIN_C0MINUS**, **SYSCTL_PIN_C0PLUS**, **SYSCTL_PIN_C0O**, **SYSCTL_PIN_C1MINUS**, **SYSCTL_PIN_C1PLUS**, **SYSCTL_PIN_C1O**, **SYSCTL_PIN_C2MINUS**, **SYSCTL_PIN_C2PLUS**, **SYSCTL_PIN_C2O**,

SYSCTL_PIN_ADC0, **SYSCTL_PIN_ADC1**, **SYSCTL_PIN_ADC2**, **SYSCTL_PIN_ADC3**, **SYSCTL_PIN_ADC4**, **SYSCTL_PIN_ADC5**, **SYSCTL_PIN_ADC6**, **SYSCTL_PIN_ADC7**, **SYSCTL_PIN_CCP0**, **SYSCTL_PIN_CCP1**, **SYSCTL_PIN_CCP2**, **SYSCTL_PIN_CCP3**, **SYSCTL_PIN_CCP4**, **SYSCTL_PIN_CCP5**, **SYSCTL_PIN_CCP6**, **SYSCTL_PIN_CCP7**, **SYSCTL_PIN_32KHZ**, or **SYSCTL_PIN_MC_FAULT0**.

Returns:

Returns **true** if the specified pin is present and **false** if it is not.

19.2.2.33 SysCtlPLLVerificationSet

Configures the PLL verification timer.

Prototype:

```
void
SysCtlPLLVerificationSet(tBoolean bEnable)
```

Parameters:

bEnable is a boolean that is **true** if the PLL verification timer should be enabled.

Description:

This function allows the PLL verification timer to be enabled or disabled. When enabled, an interrupt will be generated if the PLL ceases to operate.

The PLL verification timer is only available on Sandstorm-class devices.

Note:

The main oscillator must be enabled for this verification timer to operate as it is used to check the PLL. Also, the verification timer should be disabled while the PLL is being reconfigured via SysCtlClockSet().

Returns:

None.

19.2.2.34 SysCtlPWMClockGet

Gets the current PWM clock configuration.

Prototype:

```
unsigned long
SysCtlPWMClockGet(void)
```

Description:

This function returns the current PWM clock configuration.

Returns:

Returns the current PWM clock configuration; will be one of **SYSCTL_PWMDIV_1**, **SYSCTL_PWMDIV_2**, **SYSCTL_PWMDIV_4**, **SYSCTL_PWMDIV_8**, **SYSCTL_PWMDIV_16**, **SYSCTL_PWMDIV_32**, or **SYSCTL_PWMDIV_64**.

19.2.2.35 SysCtlPWMClockSet

Sets the PWM clock configuration.

Prototype:

```
void
SysCtlPWMClockSet(unsigned long ulConfig)
```

Parameters:

ulConfig is the configuration for the PWM clock; it must be one of **SYSCTL_PWMDIV_1**, **SYSCTL_PWMDIV_2**, **SYSCTL_PWMDIV_4**, **SYSCTL_PWMDIV_8**, **SYSCTL_PWMDIV_16**, **SYSCTL_PWMDIV_32**, or **SYSCTL_PWMDIV_64**.

Description:

This function sets the rate of the clock provided to the PWM module as a ratio of the processor clock. This clock is used by the PWM module to generate PWM signals; its rate forms the basis for all PWM signals.

Note:

The clocking of the PWM is dependent upon the system clock rate as configured by SysCtlClockSet().

Returns:

None.

19.2.2.36 SysCtlReset

Resets the device.

Prototype:

```
void
SysCtlReset(void)
```

Description:

This function will perform a software reset of the entire device. The processor and all peripherals will be reset and all device registers will return to their default values (with the exception of the reset cause register, which will maintain its current value but have the software reset bit set as well).

Returns:

This function does not return.

19.2.2.37 SysCtlResetCauseClear

Clears reset reasons.

Prototype:

```
void
SysCtlResetCauseClear(unsigned long ulCauses)
```

Parameters:

ulCauses are the reset causes to be cleared; must be a logical OR of **SYSCTL_CAUSE_LDO**, **SYSCTL_CAUSE_SW**, **SYSCTL_CAUSE_WDOG**, **SYSCTL_CAUSE_BOR**, **SYSCTL_CAUSE_POR**, and/or **SYSCTL_CAUSE_EXT**.

Description:

This function clears the specified sticky reset reasons. Once cleared, another reset for the same reason can be detected, and a reset for a different reason can be distinguished (instead of having two reset causes set). If the reset reason is used by an application, all reset causes should be cleared after they are retrieved with SysCtlResetCauseGet().

Returns:

None.

19.2.2.38 SysCtlResetCauseGet

Gets the reason for a reset.

Prototype:

```
unsigned long
SysCtlResetCauseGet(void)
```

Description:

This function will return the reason(s) for a reset. Since the reset reasons are sticky until either cleared by software or an external reset, multiple reset reasons may be returned if multiple resets have occurred. The reset reason will be a logical OR of **SYSCTL_CAUSE_LDO**, **SYSCTL_CAUSE_SW**, **SYSCTL_CAUSE_WDOG**, **SYSCTL_CAUSE_BOR**, **SYSCTL_CAUSE_POR**, and/or **SYSCTL_CAUSE_EXT**.

Returns:

Returns the reason(s) for a reset.

19.2.2.39 SysCtlSleep

Puts the processor into sleep mode.

Prototype:

```
void
SysCtlSleep(void)
```

Description:

This function places the processor into sleep mode; it will not return until the processor returns to run mode. The peripherals that are enabled via SysCtlPeripheralSleepEnable() continue to operate and can wake up the processor (if automatic clock gating is enabled with SysCtlPeripheralClockGating(), otherwise all peripherals continue to operate).

Returns:

None.

19.2.2.40 SysCtlSRAMSizeGet

Gets the size of the SRAM.

Prototype:

```
unsigned long
SysCtlSRAMSizeGet(void)
```

Description:

This function determines the size of the SRAM on the Stellaris device.

Returns:

The total number of bytes of SRAM.

19.2.2.41 SysCtlUSBPLLDisable

Powers down the USB PLL.

Prototype:

```
void
SysCtlUSBPLLDisable(void)
```

Description:

This function will disable the USB controller's PLL which is used by it's physical layer. The USB registers are still accessible, but the physical layer will no longer function.

Returns:

None.

19.2.2.42 SysCtlUSBPLLEnable

Powers up the USB PLL.

Prototype:

```
void
SysCtlUSBPLLEnable(void)
```

Description:

This function will enable the USB controller's PLL which is used by it's physical layer. This call is necessary before connecting to any external devices.

Returns:

None.

19.3 Programming Example

The following example shows how to use the SysCtl API to configure the device for normal operation.

```
//
// Configure the device to run at 20 MHz from the PLL using a 4 MHz crystal
// as the input.
//
SysCtlClockSet(SYSCTL_SYSDIV_10 | SYSCTL_USE_PLL | SYSCTL_XTAL_4MHZ |
               SYSCTL_OSC_MAIN);

//
// Enable the GPIO blocks and the SSI.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI);

//
// Enable the GPIO blocks and the SSI in sleep mode.
//
SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_GPIOB);
SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_SSI);

//
// Enable peripheral clock gating.
//
SysCtlPeripheralClockGating(true);
```

20 System Tick (SysTick)

20.1 Introduction

SysTick is a simple timer that is part of the NVIC controller in the Cortex-M3 microprocessor. Its intended purpose is to provide a periodic interrupt for a RTOS, but it can be used for other simple timing purposes.

The SysTick interrupt handler does not need to clear the SysTick interrupt source. This will be done automatically by NVIC when the SysTick interrupt handler is called.

This driver is contained in `src/systick.c`, with `src/systick.h` containing the API definitions for use by applications.

20.2 API Functions

Functions

- void SysTickDisable (void)
- void SysTickEnable (void)
- void SysTickIntDisable (void)
- void SysTickIntEnable (void)
- void SysTickIntRegister (void (*pfnHandler)(void))
- void SysTickIntUnregister (void)
- unsigned long SysTickPeriodGet (void)
- void SysTickPeriodSet (unsigned long ulPeriod)
- unsigned long SysTickValueGet (void)

20.2.1 Detailed Description

The SysTick API is fairly simple, like SysTick itself. There are functions for configuring and enabling SysTick (SysTickEnable(), SysTickDisable(), SysTickPeriodSet(), SysTickPeriodGet(), and SysTickValueGet()) and functions for dealing with an interrupt handler for SysTick (SysTickIntRegister(), SysTickIntUnregister(), SysTickIntEnable(), and SysTickIntDisable()).

20.2.2 Function Documentation

20.2.2.1 SysTickDisable

Disables the SysTick counter.

Prototype:

```
void
SysTickDisable(void)
```

Description:

This will stop the SysTick counter. If an interrupt handler has been registered, it will no longer be called until SysTick is restarted.

Returns:

None.

20.2.2.2 SysTickEnable

Enables the SysTick counter.

Prototype:

```
void
SysTickEnable(void)
```

Description:

This will start the SysTick counter. If an interrupt handler has been registered, it will be called when the SysTick counter rolls over.

Note:

Calling this function will cause the SysTick counter to (re)commence counting from its current value. The counter is not automatically reloaded with the period as specified in a previous call to SysTickPeriodSet(). If an immediate reload is required, the **NVIC_ST_CURRENT** register must be written to force this. Any write to this register clears the SysTick counter to 0 and will cause a reload with the supplied period on the next clock.

Returns:

None.

20.2.2.3 SysTickIntDisable

Disables the SysTick interrupt.

Prototype:

```
void
SysTickIntDisable(void)
```

Description:

This function will disable the SysTick interrupt, preventing it from being reflected to the processor.

Returns:

None.

20.2.2.4 SysTickIntEnable

Enables the SysTick interrupt.

Prototype:

```
void
SysTickIntEnable(void)
```

Description:

This function will enable the SysTick interrupt, allowing it to be reflected to the processor.

Note:

The SysTick interrupt handler does not need to clear the SysTick interrupt source as this is done automatically by NVIC when the interrupt handler is called.

Returns:

None.

20.2.2.5 SysTickIntRegister

Registers an interrupt handler for the SysTick interrupt.

Prototype:

```
void
SysTickIntRegister(void (*pfnHandler)(void))
```

Parameters:

pfnHandler is a pointer to the function to be called when the SysTick interrupt occurs.

Description:

This sets the handler to be called when a SysTick interrupt occurs.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

20.2.2.6 SysTickIntUnregister

Unregisters the interrupt handler for the SysTick interrupt.

Prototype:

```
void
SysTickIntUnregister(void)
```

Description:

This function will clear the handler to be called when a SysTick interrupt occurs.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:
None.

20.2.2.7 SysTickPeriodGet

Gets the period of the SysTick counter.

Prototype:
```
unsigned long
SysTickPeriodGet(void)
```

Description:
This function returns the rate at which the SysTick counter wraps; this equates to the number of processor clocks between interrupts.

Returns:
Returns the period of the SysTick counter.

20.2.2.8 SysTickPeriodSet

Sets the period of the SysTick counter.

Prototype:
```
void
SysTickPeriodSet(unsigned long ulPeriod)
```

Parameters:
ulPeriod is the number of clock ticks in each period of the SysTick counter; must be between 1 and 16,777,216, inclusive.

Description:
This function sets the rate at which the SysTick counter wraps; this equates to the number of processor clocks between interrupts.

Note:
Calling this function does not cause the SysTick counter to reload immediately. If an immediate reload is required, the **NVIC_ST_CURRENT** register must be written. Any write to this register clears the SysTick counter to 0 and will cause a reload with the *ulPeriod* supplied here on the next clock after the SysTick is enabled.

Returns:
None.

20.2.2.9 SysTickValueGet

Gets the current value of the SysTick counter.

Prototype:
```
unsigned long
SysTickValueGet(void)
```

Description:

This function returns the current value of the SysTick counter; this will be a value between the period - 1 and zero, inclusive.

Returns:

Returns the current value of the SysTick counter.

20.3 Programming Example

The following example shows how to use the SysTick API to configure the SysTick counter and read its value.

```
unsigned long ulValue;

//
// Configure and enable the SysTick counter.
//
SysTickPeriodSet(1000);
SysTickEnable();

//
// Delay for some time...
//

//
// Read the current SysTick value.
//
ulValue = SysTickValueGet();
```

21 Timer

21.1 Introduction

The timer API provides a set of functions for dealing with the timer module. Functions are provided to configure and control the timer, along with functions to modify timer/counter values, and to manage interrupt handling for the timer.

The timer module provides two 16-bit timer/counters that can be configured to operate independently as timers or event counters, or they can be configured to operate as one 32-bit timer or one 32-bit Real Time Clock (RTC). For the purpose of this API, the two timers provided by the timer are referred to as TimerA and TimerB.

When configured as either a 32-bit or 16-bit timer, a timer can be set up to run as a one-shot timer or a continuous timer. If configured as a one-shot timer, when it reaches zero the timer will cease counting. If configured as a continuous timer, when it reaches zero the timer will continue counting from a reloaded value. When configured as a 32-bit timer, the timer can also be configured to operate as an RTC. In that case, the timer expects to be driven by a 32 KHz external clock, which is divided down to produce 1 second clock ticks.

When in 16-bit mode, the timer can also be configured for event capture or as a Pulse Width Modulation (PWM) generator. When configured for event capture, the timer acts as a counter. It can be configured to either count the time between events, or it can count the events themselves. The type of event being counted can be configured as a positive edge, a negative edge, or both edges. When a timer is configured as a PWM generator, the input line used to capture events becomes an output line, and the timer is used to drive an edge-aligned pulse onto that line.

The timer module also provides the ability to control other functional parameters, such as output inversion, output triggers, and timer behavior during stalls.

Control is also provided over interrupt sources and events. Interrupts can be generated to indicate that an event has been captured, or that a certain number of events have been captured. Interrupts can also be generated when the timer has counted down to zero, or when the RTC matches a certain value.

This driver is contained in `src/timer.c`, with `src/timer.h` containing the API definitions for use by applications.

21.2 API Functions

Functions

- void TimerConfigure (unsigned long ulBase, unsigned long ulConfig)
- void TimerControlEvent (unsigned long ulBase, unsigned long ulTimer, unsigned long ulEvent)
- void TimerControlLevel (unsigned long ulBase, unsigned long ulTimer, tBoolean bInvert)

- void TimerControlStall (unsigned long ulBase, unsigned long ulTimer, tBoolean bStall)
- void TimerControlTrigger (unsigned long ulBase, unsigned long ulTimer, tBoolean bEnable)
- void TimerDisable (unsigned long ulBase, unsigned long ulTimer)
- void TimerEnable (unsigned long ulBase, unsigned long ulTimer)
- void TimerIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void TimerIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void TimerIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- void TimerIntRegister (unsigned long ulBase, unsigned long ulTimer, void (*pfnHandler)(void))
- unsigned long TimerIntStatus (unsigned long ulBase, tBoolean bMasked)
- void TimerIntUnregister (unsigned long ulBase, unsigned long ulTimer)
- unsigned long TimerLoadGet (unsigned long ulBase, unsigned long ulTimer)
- void TimerLoadSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- unsigned long TimerMatchGet (unsigned long ulBase, unsigned long ulTimer)
- void TimerMatchSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- unsigned long TimerPrescaleGet (unsigned long ulBase, unsigned long ulTimer)
- void TimerPrescaleSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- void TimerRTCDisable (unsigned long ulBase)
- void TimerRTCEnable (unsigned long ulBase)
- unsigned long TimerValueGet (unsigned long ulBase, unsigned long ulTimer)

21.2.1 Detailed Description

The timer API is broken into three groups of functions: those that deal with timer configuration and control, those that deal with timer contents, and those that deal with interrupt handling.

Timer configuration is handled by TimerConfigure(), which performs the high level setup of the timer module; that is, it is used to set up 32- or 16-bit modes, and to select between PWM, capture, and timer operations. Timer control is performed by TimerEnable(), TimerDisable(), TimerControlLevel(), TimerControlTrigger(), TimerControlEvent(), TimerControlStall(), TimerRTCEnable(), and TimerRTCDisable().

Timer content is managed with TimerLoadSet(), TimerLoadGet(), TimerPrescaleSet(), TimerPrescaleGet(), TimerMatchSet(), TimerMatchGet(), TimerPrescaleMatchSet(), TimerPrescaleMatchGet(), and TimerValueGet().

The interrupt handler for the Timer interrupt is managed with TimerIntRegister() and TimerIntUnregister(). The individual interrupt sources within the timer module are managed with TimerIntEnable(), TimerIntDisable(), TimerIntStatus(), and TimerIntClear().

The TimerQuiesce() API from previous versions of the peripheral driver library has been deprecated. SysCtlPeripheralReset() should be used instead to return the timer to its reset state.

21.2.2 Function Documentation

21.2.2.1 TimerConfigure

Configures the timer(s).

Prototype:

```
void
TimerConfigure(unsigned long ulBase,
               unsigned long ulConfig)
```

Parameters:

ulBase is the base address of the timer module.

ulConfig is the configuration for the timer.

Description:

This function configures the operating mode of the timer(s). The timer module is disabled before being configured, and is left in the disabled state. The configuration is specified in *ulConfig* as one of the following values:

- **TIMER_CFG_32_BIT_OS** - 32-bit one shot timer
- **TIMER_CFG_32_BIT_PER** - 32-bit periodic timer
- **TIMER_CFG_32_RTC** - 32-bit real time clock timer
- **TIMER_CFG_16_BIT_PAIR** - Two 16-bit timers

When configured for a pair of 16-bit timers, each timer is separately configured. The first timer is configured by setting *ulConfig* to the result of a logical OR operation between one of the following values and *ulConfig:*

- **TIMER_CFG_A_ONE_SHOT** - 16-bit one shot timer
- **TIMER_CFG_A_PERIODIC** - 16-bit periodic timer
- **TIMER_CFG_A_CAP_COUNT** - 16-bit edge count capture
- **TIMER_CFG_A_CAP_TIME** - 16-bit edge time capture
- **TIMER_CFG_A_PWM** - 16-bit PWM output

Similarly, the second timer is configured by setting *ulConfig* to the result of a logical OR operation between one of the corresponding **TIMER_CFG_B_**$*$ values and *ulConfig*.

Returns:

None.

21.2.2.2 TimerControlEvent

Controls the event type.

Prototype:

```
void
TimerControlEvent(unsigned long ulBase,
                  unsigned long ulTimer,
                  unsigned long ulEvent)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s) to be adjusted; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

ulEvent specifies the type of event; must be one of **TIMER_EVENT_POS_EDGE**, **TIMER_EVENT_NEG_EDGE**, or **TIMER_EVENT_BOTH_EDGES**.

Description:
This function sets the signal edge(s) that will trigger the timer when in capture mode.

Returns:
None.

21.2.2.3 TimerControlLevel

Controls the output level.

Prototype:
```
void
TimerControlLevel(unsigned long ulBase,
                  unsigned long ulTimer,
                  tBoolean bInvert)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.
bInvert specifies the output level.

Description:
This function sets the PWM output level for the specified timer. If the *bInvert* parameter is **true**, then the timer's output will be made active low; otherwise, it will be made active high.

Returns:
None.

21.2.2.4 TimerControlStall

Controls the stall handling.

Prototype:
```
void
TimerControlStall(unsigned long ulBase,
                  unsigned long ulTimer,
                  tBoolean bStall)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer(s) to be adjusted; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.
bStall specifies the response to a stall signal.

Description:
This function controls the stall response for the specified timer. If the *bStall* parameter is **true**, then the timer will stop counting if the processor enters debug mode; otherwise the timer will keep running while in debug mode.

Returns:
None.

21.2.2.5 TimerControlTrigger

Enables or disables the trigger output.

Prototype:

```
void
TimerControlTrigger(unsigned long ulBase,
                    unsigned long ulTimer,
                    tBoolean bEnable)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

bEnable specifies the desired trigger state.

Description:

This function controls the trigger output for the specified timer. If the *bEnable* parameter is **true**, then the timer's output trigger is enabled; otherwise it is disabled.

Returns:

None.

21.2.2.6 TimerDisable

Disables the timer(s).

Prototype:

```
void
TimerDisable(unsigned long ulBase,
             unsigned long ulTimer)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s) to disable; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

Description:

This will disable operation of the timer module.

Returns:

None.

21.2.2.7 TimerEnable

Enables the timer(s).

Prototype:

```
void
TimerEnable(unsigned long ulBase,
            unsigned long ulTimer)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s) to enable; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

Description:

This will enable operation of the timer module. The timer must be configured before it is enabled.

Returns:

None.

21.2.2.8 TimerIntClear

Clears timer interrupt sources.

Prototype:

```
void
TimerIntClear(unsigned long ulBase,
              unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the timer module.

ulIntFlags is a bit mask of the interrupt sources to be cleared.

Description:

The specified timer interrupt sources are cleared, so that they no longer assert. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to TimerIntEnable().

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

21.2.2.9 TimerIntDisable

Disables individual timer interrupt sources.

Prototype:

```
void
TimerIntDisable(unsigned long ulBase,
                unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the timer module.

ulIntFlags is the bit mask of the interrupt sources to be disabled.

Description:

Disables the indicated timer interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to TimerIntEnable().

Returns:

None.

21.2.2.10 TimerIntEnable

Enables individual timer interrupt sources.

Prototype:

```
void
TimerIntEnable(unsigned long ulBase,
               unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the timer module.

ulIntFlags is the bit mask of the interrupt sources to be enabled.

Description:

Enables the indicated timer interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ulIntFlags* parameter must be the logical OR of any combination of the following:

- **TIMER_CAPB_EVENT** - Capture B event interrupt
- **TIMER_CAPB_MATCH** - Capture B match interrupt
- **TIMER_TIMB_TIMEOUT** - Timer B timeout interrupt
- **TIMER_RTC_MATCH** - RTC interrupt mask
- **TIMER_CAPA_EVENT** - Capture A event interrupt
- **TIMER_CAPA_MATCH** - Capture A match interrupt
- **TIMER_TIMA_TIMEOUT** - Timer A timeout interrupt

Returns:

None.

21.2.2.11 TimerIntRegister

Registers an interrupt handler for the timer interrupt.

Prototype:

```
void
TimerIntRegister(unsigned long ulBase,
```

```
                        unsigned long ulTimer,
                        void (*pfnHandler)(void))
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s); must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

pfnHandler is a pointer to the function to be called when the timer interrupt occurs.

Description:

This sets the handler to be called when a timer interrupt occurs. This will enable the global interrupt in the interrupt controller; specific timer interrupts must be enabled via TimerIntEnable(). It is the interrupt handler's responsibility to clear the interrupt source via TimerIntClear().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

21.2.2.12 TimerIntStatus

Gets the current interrupt status.

Prototype:

```
unsigned long
TimerIntStatus(unsigned long ulBase,
               tBoolean bMasked)
```

Parameters:

ulBase is the base address of the timer module.

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This returns the interrupt status for the timer module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

The current interrupt status, enumerated as a bit field of values described in TimerIntEnable().

21.2.2.13 TimerIntUnregister

Unregisters an interrupt handler for the timer interrupt.

Prototype:

```
void
TimerIntUnregister(unsigned long ulBase,
                   unsigned long ulTimer)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s); must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

Description:
This function will clear the handler to be called when a timer interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

21.2.2.14 TimerLoadGet

Gets the timer load value.

Prototype:
```
unsigned long
TimerLoadGet(unsigned long ulBase,
             unsigned long ulTimer)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.

Description:
This function gets the currently programmed interval load value for the specified timer.

Returns:
Returns the load value for the timer.

21.2.2.15 TimerLoadSet

Sets the timer load value.

Prototype:
```
void
TimerLoadSet(unsigned long ulBase,
             unsigned long ulTimer,
             unsigned long ulValue)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.
ulValue is the load value.

Description:
This function sets the timer load value; if the timer is running then the value will be immediately loaded into the timer.

Returns:
None.

21.2.2.16 TimerMatchGet

Gets the timer match value.

Prototype:
```
unsigned long
TimerMatchGet(unsigned long ulBase,
              unsigned long ulTimer)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.

Description:
This function gets the match value for the specified timer.

Returns:
Returns the match value for the timer.

21.2.2.17 TimerMatchSet

Sets the timer match value.

Prototype:
```
void
TimerMatchSet(unsigned long ulBase,
              unsigned long ulTimer,
              unsigned long ulValue)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.
ulValue is the match value.

Description:
This function sets the match value for a timer. This is used in capture count mode to determine when to interrupt the processor and in PWM mode to determine the duty cycle of the output signal.

Returns:
None.

21.2.2.18 TimerPrescaleGet

Get the timer prescale value.

Prototype:

```
unsigned long
TimerPrescaleGet(unsigned long ulBase,
                 unsigned long ulTimer)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer; must be one of **TIMER_A** or **TIMER_B**.

Description:

This function gets the value of the input clock prescaler. The prescaler is only operational when in 16-bit mode and is used to extend the range of the 16-bit timer modes.

Returns:

The value of the timer prescaler.

21.2.2.19 TimerPrescaleSet

Set the timer prescale value.

Prototype:

```
void
TimerPrescaleSet(unsigned long ulBase,
                 unsigned long ulTimer,
                 unsigned long ulValue)
```

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

ulValue is the timer prescale value; must be between 0 and 255, inclusive.

Description:

This function sets the value of the input clock prescaler. The prescaler is only operational when in 16-bit mode and is used to extend the range of the 16-bit timer modes.

Returns:

None.

21.2.2.20 TimerRTCDisable

Disable RTC counting.

Prototype:

```
void
TimerRTCDisable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the timer module.

Description:
This function causes the timer to stop counting when in RTC mode.

Returns:
None.

21.2.2.21 TimerRTCEnable

Enable RTC counting.

Prototype:
```
void
TimerRTCEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the timer module.

Description:
This function causes the timer to start counting when in RTC mode. If not configured for RTC mode, this will do nothing.

Returns:
None.

21.2.2.22 TimerValueGet

Gets the current timer value.

Prototype:
```
unsigned long
TimerValueGet(unsigned long ulBase,
              unsigned long ulTimer)
```

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.

Description:
This function reads the current value of the specified timer.

Returns:
Returns the current value of the timer.

21.3 Programming Example

The following example shows how to use the timer API to configure the timer as a 16-bit one shot timer and a 16-bit edge capture counter.

```
//
// Configure TimerA as a 16-bit one shot timer, and TimerB as a 16-bit edge
// capture counter.
//
TimerConfigure(TIMER0_BASE, (TIMER_CFG_16_BIT_PAIR | TIMER_CFG_A_ONE_SHOT |
                             TIMER_CFG_B_CAP_COUNT));

//
// Configure the counter (TimerB) to count both edges.
//
TimerControlEvent(TIMER0_BASE, TIMER_B, TIMER_EVENT_BOTH_EDGES);

//
// Enable the timers.
//
TimerEnable(TIMER0_BASE, TIMER_BOTH);
```

22 UART

22.1 Introduction

The Universal Asynchronous Receiver/Transmitter (UART) API provides a set of functions for using the Stellaris UART modules. Functions are provided to configure and control the UART modules, to send and receive data, and to manage interrupts for the UART modules.

The Stellaris UART performs the functions of parallel-to-serial and serial-to-parallel conversions. It is very similar in functionality to a 16C550 UART, but is not register-compatible.

Some of the features of the Stellaris UART are:

- A 16x12 bit receive FIFO and a 16x8 bit transmit FIFO.
- Programmable baud rate generator.
- Automatic generation and stripping of start, stop, and parity bits.
- Line break generation and detection.
- Programmable serial interface
  - 5, 6, 7, or 8 data bits
  - even, odd, stick, or no parity bit generation and detection
  - 1 or 2 stop bit generation
  - baud rate generation, from DC to processor clock/16
- IrDA serial-IR (SIR) encoder/decoder.
- DMA interface

This driver is contained in `src/uart.c`, with `src/uart.h` containing the API definitions for use by applications.

22.2 API Functions

Functions

- void UARTBreakCtl (unsigned long ulBase, tBoolean bBreakState)
- tBoolean UARTBusy (unsigned long ulBase)
- long UARTCharGet (unsigned long ulBase)
- long UARTCharGetNonBlocking (unsigned long ulBase)
- void UARTCharPut (unsigned long ulBase, unsigned char ucData)
- tBoolean UARTCharPutNonBlocking (unsigned long ulBase, unsigned char ucData)
- tBoolean UARTCharsAvail (unsigned long ulBase)
- void UARTConfigGetExpClk (unsigned long ulBase, unsigned long ulUARTClk, unsigned long *pulBaud, unsigned long *pulConfig)

- void UARTConfigSetExpClk (unsigned long ulBase, unsigned long ulUARTClk, unsigned long ulBaud, unsigned long ulConfig)
- void UARTDisable (unsigned long ulBase)
- void UARTDisableSIR (unsigned long ulBase)
- void UARTDMADisable (unsigned long ulBase, unsigned long ulDMAFlags)
- void UARTDMAEnable (unsigned long ulBase, unsigned long ulDMAFlags)
- void UARTEnable (unsigned long ulBase)
- void UARTEnableSIR (unsigned long ulBase, tBoolean bLowPower)
- void UARTFIFOLevelGet (unsigned long ulBase, unsigned long *pulTxLevel, unsigned long *pulRxLevel)
- void UARTFIFOLevelSet (unsigned long ulBase, unsigned long ulTxLevel, unsigned long ulRxLevel)
- void UARTIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void UARTIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void UARTIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- void UARTIntRegister (unsigned long ulBase, void (*pfnHandler)(void))
- unsigned long UARTIntStatus (unsigned long ulBase, tBoolean bMasked)
- void UARTIntUnregister (unsigned long ulBase)
- unsigned long UARTParityModeGet (unsigned long ulBase)
- void UARTParityModeSet (unsigned long ulBase, unsigned long ulParity)
- void UARTRxErrorClear (unsigned long ulBase)
- unsigned long UARTRxErrorGet (unsigned long ulBase)
- tBoolean UARTSpaceAvail (unsigned long ulBase)

22.2.1 Detailed Description

The UART API provides the set of functions required to implement an interrupt driven UART driver. These functions may be used to control any of the available UART ports on a Stellaris microcontroller, and can be used with one port without causing conflicts with the other port.

The UART API is broken into three groups of functions: those that deal with configuration and control of the UART modules, those used to send and receive data, and those that deal with interrupt handling.

Configuration and control of the UART are handled by the UARTConfigGetExpClk(), UARTConfigSetExpClk(), UARTDisable(), UARTEnable(), UARTParityModeGet(), and UARTParityModeSet() functions. The DMA interface can be enabled or disabled by the UARTDMAEnable() and UARTDMADisable() functions.

Sending and receiving data via the UART is handled by the UARTCharGet(), UARTCharGetNonBlocking(), UARTCharPut(), UARTCharPutNonBlocking(), UARTBreakCtl(), UARTCharsAvail(), and UARTSpaceAvail() functions.

Managing the UART interrupts is handled by the UARTIntClear(), UARTIntDisable(), UARTIntEnable(), UARTIntRegister(), UARTIntStatus(), and UARTIntUnregister() functions.

The UARTConfigSet(), UARTConfigGet(), UARTCharNonBlockingGet(), and UARTCharNonBlockingPut() APIs from previous versions of the peripheral driver library have been replaced by the UARTConfigSetExpClk(), UARTConfigGetExpClk(), UARTCharGetNonBlocking(), and UARTCharPutNonBlocking() APIs, respectively. Macros have been provided in `uart.h` to map the old APIs to the new APIs, allowing existing applications to link and run with the new APIs. It is recommended that new applications utilize the new APIs in favor of the old ones.

22.2.2 Function Documentation

22.2.2.1 UARTBreakCtl

Causes a BREAK to be sent.

Prototype:

```
void
UARTBreakCtl(unsigned long ulBase,
             tBoolean bBreakState)
```

Parameters:

ulBase is the base address of the UART port.

bBreakState controls the output level.

Description:

Calling this function with *bBreakState* set to **true** will assert a break condition on the UART. Calling this function with *bBreakState* set to **false** will remove the break condition. For proper transmission of a break command, the break must be asserted for at least two complete frames.

Returns:

None.

22.2.2.2 UARTBusy

Determines whether the UART transmitter is busy or not.

Prototype:

```
tBoolean
UARTBusy(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

Description:

Allows the caller to determine whether all transmitted bytes have cleared the transmitter hardware. If **false** is returned, the transmit FIFO is empty and all bits of the last transmitted character, including all stop bits, have left the hardware shift register.

Returns:

Returns **true** if the UART is transmitting or **false** if all transmissions are complete.

22.2.2.3 UARTCharGet

Waits for a character from the specified port.

Prototype:

```
long
UARTCharGet(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the UART port.

Description:
Gets a character from the receive FIFO for the specified port. If there are no characters available, this function will wait until a character is received before returning.

Returns:
Returns the character read from the specified port, cast as an *long*.

22.2.2.4 UARTCharGetNonBlocking

Receives a character from the specified port.

Prototype:
```
long
UARTCharGetNonBlocking(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the UART port.

Description:
Gets a character from the receive FIFO for the specified port.

This function replaces the original UARTCharNonBlockingGet() API and performs the same actions. A macro is provided in `uart.h` to map the original API to this API.

Returns:
Returns the character read from the specified port, cast as a *long*. A **-1** will be returned if there are no characters present in the receive FIFO. The UARTCharsAvail() function should be called before attempting to call this function.

22.2.2.5 UARTCharPut

Waits to send a character from the specified port.

Prototype:
```
void
UARTCharPut(unsigned long ulBase,
            unsigned char ucData)
```

Parameters:
ulBase is the base address of the UART port.
ucData is the character to be transmitted.

Description:
Sends the character *ucData* to the transmit FIFO for the specified port. If there is no space available in the transmit FIFO, this function will wait until there is space available before returning.

Returns:
None.

22.2.2.6 UARTCharPutNonBlocking

Sends a character to the specified port.

Prototype:

```
tBoolean
UARTCharPutNonBlocking(unsigned long ulBase,
                       unsigned char ucData)
```

Parameters:

ulBase is the base address of the UART port.

ucData is the character to be transmitted.

Description:

Writes the character *ucData* to the transmit FIFO for the specified port. This function does not block, so if there is no space available, then a **false** is returned, and the application will have to retry the function later.

This function replaces the original UARTCharNonBlockingPut() API and performs the same actions. A macro is provided in `uart.h` to map the original API to this API.

Returns:

Returns **true** if the character was successfully placed in the transmit FIFO, and **false** if there was no space available in the transmit FIFO.

22.2.2.7 UARTCharsAvail

Determines if there are any characters in the receive FIFO.

Prototype:

```
tBoolean
UARTCharsAvail(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

Description:

This function returns a flag indicating whether or not there is data available in the receive FIFO.

Returns:

Returns **true** if there is data in the receive FIFO, and **false** if there is no data in the receive FIFO.

22.2.2.8 UARTConfigGetExpClk

Gets the current configuration of a UART.

Prototype:

```
void
UARTConfigGetExpClk(unsigned long ulBase,
                    unsigned long ulUARTClk,
```

```
                        unsigned long *pulBaud,
                        unsigned long *pulConfig)
```

Parameters:

ulBase is the base address of the UART port.

ulUARTClk is the rate of the clock supplied to the UART module.

pulBaud is a pointer to storage for the baud rate.

pulConfig is a pointer to storage for the data format.

Description:

The baud rate and data format for the UART is determined, given an explicitly provided peripheral clock (hence the ExpClk suffix). The returned baud rate is the actual baud rate; it may not be the exact baud rate requested or an "official" baud rate. The data format returned in *pulConfig* is enumerated the same as the *ulConfig* parameter of UARTConfigSetExpClk().

The peripheral clock will be the same as the processor clock. This will be the value returned by SysCtlClockGet(), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to SysCtlClockGet()).

This function replaces the original UARTConfigGet() API and performs the same actions. A macro is provided in `uart.h` to map the original API to this API.

Returns:

None.

22.2.2.9 UARTConfigSetExpClk

Sets the configuration of a UART.

Prototype:

```
void
UARTConfigSetExpClk(unsigned long ulBase,
                    unsigned long ulUARTClk,
                    unsigned long ulBaud,
                    unsigned long ulConfig)
```

Parameters:

ulBase is the base address of the UART port.

ulUARTClk is the rate of the clock supplied to the UART module.

ulBaud is the desired baud rate.

ulConfig is the data format for the port (number of data bits, number of stop bits, and parity).

Description:

This function will configure the UART for operation in the specified data format. The baud rate is provided in the *ulBaud* parameter and the data format in the *ulConfig* parameter.

The *ulConfig* parameter is the logical OR of three values: the number of data bits, the number of stop bits, and the parity. **UART_CONFIG_WLEN_8**, **UART_CONFIG_WLEN_7**, **UART_CONFIG_WLEN_6**, and **UART_CONFIG_WLEN_5** select from eight to five data bits per byte (respectively). **UART_CONFIG_STOP_ONE** and **UART_CONFIG_STOP_TWO** select one or two stop bits (respectively). **UART_CONFIG_PAR_NONE**, **UART_CONFIG_PAR_EVEN**, **UART_CONFIG_PAR_ODD**, **UART_CONFIG_PAR_ONE**,

and **UART_CONFIG_PAR_ZERO** select the parity mode (no parity bit, even parity bit, odd parity bit, parity bit always one, and parity bit always zero, respectively).

The peripheral clock will be the same as the processor clock. This will be the value returned by SysCtlClockGet(), or it can be explicitly hard coded if it is constant and known (to save the code/execution overhead of a call to SysCtlClockGet()).

This function replaces the original UARTConfigSet() API and performs the same actions. A macro is provided in `uart.h` to map the original API to this API.

Returns:
None.

22.2.2.10 UARTDisable

Disables transmitting and receiving.

Prototype:
```
void
UARTDisable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the UART port.

Description:
Clears the UARTEN, TXE, and RXE bits, then waits for the end of transmission of the current character, and flushes the transmit FIFO.

Returns:
None.

22.2.2.11 UARTDisableSIR

Disables SIR (IrDA) mode on the specified UART.

Prototype:
```
void
UARTDisableSIR(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the UART port.

Description:
Clears the SIREN (IrDA) and SIRLP (Low Power) bits.

Note:
SIR (IrDA) operation is not supported on Sandstorm-class devices.

Returns:
None.

22.2.2.12 UARTDMADisable

Disable UART DMA operation.

Prototype:

```
void
UARTDMADisable(unsigned long ulBase,
               unsigned long ulDMAFlags)
```

Parameters:

ulBase is the base address of the UART port.

ulDMAFlags is a bit mask of the DMA features to disable.

Description:

This function is used to disable UART DMA features that were enabled by UARTDMAEnable(). The specified UART DMA features are disabled. The *ulDMAFlags* parameter is the logical OR of any of the following values:

- UART_DMA_RX - disable DMA for receive
- UART_DMA_TX - disable DMA for transmit
- UART_DMA_ERR_RXSTOP - do not disable DMA receive on UART error

Returns:

None.

22.2.2.13 UARTDMAEnable

Enable UART DMA operation.

Prototype:

```
void
UARTDMAEnable(unsigned long ulBase,
              unsigned long ulDMAFlags)
```

Parameters:

ulBase is the base address of the UART port.

ulDMAFlags is a bit mask of the DMA features to enable.

Description:

The specified UART DMA features are enabled. The UART can be configured to use DMA for transmit or receive, and to disable receive if an error occurs. The *ulDMAFlags* parameter is the logical OR of any of the following values:

- UART_DMA_RX - enable DMA for receive
- UART_DMA_TX - enable DMA for transmit
- UART_DMA_ERR_RXSTOP - disable DMA receive on UART error

Note:

The uDMA controller must also be set up before DMA can be used with the UART.

Returns:

None.

22.2.2.14 UARTEnable

Enables transmitting and receiving.

Prototype:

```
void
UARTEnable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

Description:

Sets the UARTEN, TXE, and RXE bits, and enables the transmit and receive FIFOs.

Returns:

None.

22.2.2.15 UARTEnableSIR

Enables SIR (IrDA) mode on the specified UART.

Prototype:

```
void
UARTEnableSIR(unsigned long ulBase,
              tBoolean bLowPower)
```

Parameters:

ulBase is the base address of the UART port.

bLowPower indicates if SIR Low Power Mode is to be used.

Description:

Enables the SIREN control bit for IrDA mode on the UART. If the *bLowPower* flag is set, then SIRLP bit will also be set.

Note:

SIR (IrDA) operation is not supported on Sandstorm-class devices.

Returns:

None.

22.2.2.16 UARTFIFOLevelGet

Gets the FIFO level at which interrupts are generated.

Prototype:

```
void
UARTFIFOLevelGet(unsigned long ulBase,
                 unsigned long *pulTxLevel,
                 unsigned long *pulRxLevel)
```

Parameters:

ulBase is the base address of the UART port.

pulTxLevel is a pointer to storage for the transmit FIFO level, returned as one of **UART_FIFO_TX1_8**, **UART_FIFO_TX2_8**, **UART_FIFO_TX4_8**, **UART_FIFO_TX6_8**, or UART_FIFO_TX7_8.

pulRxLevel is a pointer to storage for the receive FIFO level, returned as one of **UART_FIFO_RX1_8**, **UART_FIFO_RX2_8**, **UART_FIFO_RX4_8**, **UART_FIFO_RX6_8**, or **UART_FIFO_RX7_8**.

Description:

This function gets the FIFO level at which transmit and receive interrupts will be generated.

Returns:

None.

22.2.2.17 UARTFIFOLevelSet

Sets the FIFO level at which interrupts are generated.

Prototype:

```
void
UARTFIFOLevelSet(unsigned long ulBase,
                 unsigned long ulTxLevel,
                 unsigned long ulRxLevel)
```

Parameters:

ulBase is the base address of the UART port.

ulTxLevel is the transmit FIFO interrupt level, specified as one of **UART_FIFO_TX1_8**, **UART_FIFO_TX2_8**, **UART_FIFO_TX4_8**, **UART_FIFO_TX6_8**, or **UART_FIFO_TX7_8**.

ulRxLevel is the receive FIFO interrupt level, specified as one of **UART_FIFO_RX1_8**, **UART_FIFO_RX2_8**, **UART_FIFO_RX4_8**, **UART_FIFO_RX6_8**, or **UART_FIFO_RX7_8**.

Description:

This function sets the FIFO level at which transmit and receive interrupts will be generated.

Returns:

None.

22.2.2.18 UARTIntClear

Clears UART interrupt sources.

Prototype:

```
void
UARTIntClear(unsigned long ulBase,
             unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the UART port.

ulIntFlags is a bit mask of the interrupt sources to be cleared.

Description:

The specified UART interrupt sources are cleared, so that they no longer assert. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to UARTIntEnable().

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

22.2.2.19 UARTIntDisable

Disables individual UART interrupt sources.

Prototype:

```
void
UARTIntDisable(unsigned long ulBase,
               unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the UART port.

ulIntFlags is the bit mask of the interrupt sources to be disabled.

Description:

Disables the indicated UART interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to UARTIntEnable().

Returns:

None.

22.2.2.20 UARTIntEnable

Enables individual UART interrupt sources.

Prototype:

```
void
UARTIntEnable(unsigned long ulBase,
              unsigned long ulIntFlags)
```

Parameters:

ulBase is the base address of the UART port.

ulIntFlags is the bit mask of the interrupt sources to be enabled.

Description:
Enables the indicated UART interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ulIntFlags* parameter is the logical OR of any of the following:

- **UART_INT_OE** - Overrun Error interrupt
- **UART_INT_BE** - Break Error interrupt
- **UART_INT_PE** - Parity Error interrupt
- **UART_INT_FE** - Framing Error interrupt
- **UART_INT_RT** - Receive Timeout interrupt
- **UART_INT_TX** - Transmit interrupt
- **UART_INT_RX** - Receive interrupt

Returns:
None.

22.2.2.21 UARTIntRegister

Registers an interrupt handler for a UART interrupt.

Prototype:
```
void
UARTIntRegister(unsigned long ulBase,
                void (*pfnHandler)(void))
```

Parameters:
ulBase is the base address of the UART port.
pfnHandler is a pointer to the function to be called when the UART interrupt occurs.

Description:
This function does the actual registering of the interrupt handler. This will enable the global interrupt in the interrupt controller; specific UART interrupts must be enabled via UARTIntEnable(). It is the interrupt handler's responsibility to clear the interrupt source.

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

22.2.2.22 UARTIntStatus

Gets the current interrupt status.

Prototype:
```
unsigned long
UARTIntStatus(unsigned long ulBase,
              tBoolean bMasked)
```

Parameters:

ulBase is the base address of the UART port.

bMasked is false if the raw interrupt status is required and true if the masked interrupt status is required.

Description:

This returns the interrupt status for the specified UART. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status, enumerated as a bit field of values described in UARTIntEnable().

22.2.2.23 UARTIntUnregister

Unregisters an interrupt handler for a UART interrupt.

Prototype:

```
void
UARTIntUnregister(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

Description:

This function does the actual unregistering of the interrupt handler. It will clear the handler to be called when a UART interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

22.2.2.24 UARTParityModeGet

Gets the type of parity currently being used.

Prototype:

```
unsigned long
UARTParityModeGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

Description:

This function gets the type of parity used for transmitting data, and expected when receiving data.

Returns:

Returns the current parity settings, specified as one of **UART_CONFIG_PAR_NONE**, **UART_CONFIG_PAR_EVEN**, **UART_CONFIG_PAR_ODD**, **UART_CONFIG_PAR_ONE**, or **UART_CONFIG_PAR_ZERO**.

22.2.2.25 UARTParityModeSet

Sets the type of parity.

Prototype:

```
void
UARTParityModeSet(unsigned long ulBase,
                  unsigned long ulParity)
```

Parameters:

ulBase is the base address of the UART port.

ulParity specifies the type of parity to use.

Description:

Sets the type of parity to use for transmitting and expect when receiving. The *ulParity* parameter must be one of **UART_CONFIG_PAR_NONE**, **UART_CONFIG_PAR_EVEN**, **UART_CONFIG_PAR_ODD**, **UART_CONFIG_PAR_ONE**, or **UART_CONFIG_PAR_ZERO**. The last two allow direct control of the parity bit; it will always be either be one or zero based on the mode.

Returns:

None.

22.2.2.26 UARTRxErrorClear

Clears all reported receiver errors.

Prototype:

```
void
UARTRxErrorClear(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

Description:

This function is used to clear all receiver error conditions reported via UARTRxErrorGet(). If using the overrun, framing error, parity error or break interrupts, this function must be called after clearing the interrupt to ensure that later errors of the same type trigger another interrupt.

Returns:

None.

22.2.2.27 UARTRxErrorGet

Gets current receiver errors.

Prototype:

```
unsigned long
UARTRxErrorGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

Description:

This function returns the current state of each of the 4 receiver error sources. The returned errors are equivalent to the four error bits returned via the previous call to UARTCharGet() or UARTCharGetNonBlocking() with the exception that the overrun error is set immediately the overrun occurs rather than when a character is next read.

Returns:

Returns a logical OR combination of the receiver error flags, **UART_RXERROR_FRAMING**, **UART_RXERROR_PARITY**, **UART_RXERROR_BREAK** and **UART_RXERROR_OVERRUN**.

22.2.2.28 UARTSpaceAvail

Determines if there is any space in the transmit FIFO.

Prototype:

```
tBoolean
UARTSpaceAvail(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the UART port.

Description:

This function returns a flag indicating whether or not there is space available in the transmit FIFO.

Returns:

Returns **true** if there is space available in the transmit FIFO, and **false** if there is no space available in the transmit FIFO.

22.3 Programming Example

The following example shows how to use the UART API to initialize the UART, transmit characters, and receive characters.

```
//
// Initialize the UART.  Set the baud rate, number of data bits, turn off
// parity, number of stop bits, and stick mode.
//
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 38400,
                    (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
```

```
                      UART_CONFIG_PAR_NONE));

//
// Enable the UART.
//
UARTEnable(UART0_BASE);

//
// Check for characters.  This will spin here until a character is placed
// into the receive FIFO.
//
while(!UARTCharsAvail(UART0_BASE))
{
}

//
// Get the character(s) in the receive FIFO.
//
while(UARTCharGetNonBlocking(UART0_BASE))
{
}

//
// Put a character in the output buffer.
//
UARTCharPut(UART0_BASE, 'c'));

//
// Disable the UART.
//
UARTDisable(UART0_BASE);
```

23 uDMA Controller

23.1 Introduction

The microDMA (uDMA) API provides functions to configure the Stellaris uDMA (Direct Memory Access) controller. The uDMA controller is designed to work with the the ARM Cortex-M3 processor and provides an efficient and low-overhead means of transferring blocks of data in the system.

The uDMA controller has the following features:

- dedicated channels for supported peripherals
- one channel each for receive and transmit for devices with receive and transmit paths
- dedicated channel for software initiated data transfers
- channels can be independently configured and operated
- an arbitration scheme that is configurable per channel
- two levels of priority
- subordinate to Cortex-M3 processor bus usage
- data sizes of 8, 16, or 32 bits
- address increment of byte, half-word, word, or none
- maskable device requests
- optional software initiated transfers on any channel
- interrupt on transfer completion

The uDMA controller supports several different transfer modes, allowing for complex transfer schemes. The following transfer modes are provided:

- **Basic** mode performs a simple transfer when request is asserted by a device. This is appropriate to use with peripherals where the peripheral asserts the request line whenever data should be transferred. The transfer will stop if request is de-asserted, even if the transfer is not complete.

- **Auto-request** mode performs a simple transfer that is started by a request, but will always complete the entire transfer, even if request is de-asserted. This is appropriate to use with software initiated transfers.

- **Ping-Pong** mode is used to transfer data to or from two buffers, switching from one buffer to the other as each buffer fills. This mode is appropriate to use with peripherals as a way to ensure a continuous flow of data to or from the peripheral. However, it is more complex to set up and requires code to manage the ping-pong buffers in the interrupt handler.

- **Memory scatter/gather** mode is a complex mode that provides a way to set up a list of transfer "tasks" for the uDMA controller. Blocks of data can be transferred to and from arbitrary locations in memory.

- **Peripheral scatter/gather** mode is similar to memory scatter/gather mode except that it is controlled by a peripheral request.

Detailed explanation of the various transfer modes is beyond the scope of this document. Please refer to the device data sheet for more information on the operation of the uDMA controller.

The naming convention for the microDMA controller is to use the Greek letter "mu" to represent "micro". For the purposes of this document, and in the software library function names, a lower case "u" will be used in place of "mu" when the controller is referred to as "uDMA".

This driver is contained in `src/udma.c`, with `src/udma.h` containing the API definitions for use by applications.

23.2 API Functions

Functions

- void uDMAChannelAttributeDisable (unsigned long ulChannel, unsigned long ulAttr)
- void uDMAChannelAttributeEnable (unsigned long ulChannel, unsigned long ulAttr)
- unsigned long uDMAChannelAttributeGet (unsigned long ulChannel)
- void uDMAChannelControlSet (unsigned long ulChannel, unsigned long ulControl)
- void uDMAChannelDisable (unsigned long ulChannel)
- void uDMAChannelEnable (unsigned long ulChannel)
- tBoolean uDMAChannelIsEnabled (unsigned long ulChannel)
- unsigned long uDMAChannelModeGet (unsigned long ulChannel)
- void uDMAChannelRequest (unsigned long ulChannel)
- unsigned long uDMAChannelSizeGet (unsigned long ulChannel)
- void uDMAChannelTransferSet (unsigned long ulChannel, unsigned long ulMode, void *pvSrcAddr, void *pvDstAddr, unsigned long ulTransferSize)
- void * uDMAControlBaseGet (void)
- void uDMAControlBaseSet (void *pControlTable)
- void uDMADisable (void)
- void uDMAEnable (void)
- void uDMAErrorStatusClear (void)
- unsigned long uDMAErrorStatusGet (void)
- void uDMAIntRegister (unsigned long ulIntChannel, void (*pfnHandler)(void))
- void uDMAIntUnregister (unsigned long ulIntChannel)

23.2.1 Detailed Description

The uDMA APIs provide a means to enable and configure the Stellaris microDMA controller to perform DMA transfers.

The general order of function calls to set up and perform a uDMA transfer is the following:

- uDMAEnable() is called once to enable the controller.
- uDMAControlBaseSet() is called once to set the channel control table.

- uDMAChannelAttributeEnable() is called once or infrequently to configure the behavior of the channel.
- uDMAChannelControlSet() is used to set up characteristics of the data transfer. It only needs to be called once if the nature of the data transfer does not change.
- uDMAChannelTransferSet() is used to set the buffer pointers and size for a transfer. It is called before each new transfer.
- uDMAChannelEnable() enables a channel to perform data transfers.
- uDMAChannelRequest() is used to initiate a software based transfer. This is normally not used for peripheral based transfers.

In order to use the uDMA controller, you must first enable it by calling uDMAEnable(). You can later disable it, if no longer needed, by calling uDMADisable().

Once the uDMA controller is enabled, you must tell it where to find the channel control structures in system memory. This is done by using the function uDMAControlBaseSet() and passing a pointer to the base of the channel control structure. The control structure must be allocated by the application. One way to do this is to declare an array of data type `char` or `unsigned char`. In order to support all channels and transfer modes, the control table array should be 1024 bytes, but it can be fewer depending on transfer modes used and number of channels actually used.

Note:
The control table must be aligned on a 1024 byte boundary.

The uDMA controller supports multiple channels. Each channel has a set of attribute flags to control certain uDMA features and channel behavior. The attribute flags are set with the function uDMAChannelAttributeEnable() and cleared with uDMAChannelAttributeDisable(). The setting of the channel attribute flags can be queried by using the function uDMAChannelAttributeGet().

Next, the control parameters of the DMA transfer must be set. These parameters control the size and address increment of the data items to be transferred. The function uDMAChannelControlSet() is used to set up these control parameters.

All of the functions mentioned so far are used only once or infrequently to set up the uDMA channel and transfer. In order to set the transfer addresses, transfer size, and transfer mode, use the function uDMAChannelTransferSet(). This function must be called for each new transfer. Once everything is set up, then channel is enabled by calling uDMAChannelEnable(), which must be done before each new transfer. The uDMA controller will automatically disable the channel at the completion of a transfer. A channel can be manually disabled by using uDMAChannelDisable().

There are additional functions that can be used to query the status of a channel, either from an interrupt handler or in polling fashion. The function uDMAChannelSizeGet() is used to find the amount of data remaining to transfer on a channel. This will be zero when a transfer is complete. The function uDMAChannelModeGet() can be used to find the transfer mode of a uDMA channel. This is usually used to see if the mode indicates stopped which means that a transfer has completed on a channel that was previously running. The function uDMAChannelIsEnabled() can be used to determine if a particular channel is enabled.

If the application is using run-time interrupt registration (see IntRegister()), then the function uDMAIntRegister() can be used to install an interrupt handler for the uDMA controller. This function will also enable the interrupt on the system interrupt controller. If compile-time interrupt registration is used, then call the function IntEnable() to enable uDMA interrupts. When an interrupt handler has been installed with uDMAIntRegister(), it can be removed by calling uDMAIntUnregister().

This interrupt handler is only for software initiated transfers or errors. uDMA interrupts for a peripheral occur on the peripheral's dedicated interrupt channel, and should be handled by the peripheral

interrupt handler. It is not necessary to acknowledge or clear uDMA interrupt sources. They are cleared automatically when they are serviced.

The uDMA interrupt handler should use the function uDMAErrorStatusGet() to test if a uDMA error occurred. If so, the interrupt must be cleared by calling uDMAErrorStatusClear().

Note:
Many of the API functions take a channel parameter that includes the logical OR of one of the values **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose the primary or alternate control structure. For Basic and Auto transfer modes, only the primary control structure is needed. The alternate contol structure is only needed for complex transfer modes of Ping-pong or Scatter/gather. Refer to the device data sheet for detailed information about transfer modes.

23.2.2 Function Documentation

23.2.2.1 uDMAChannelAttributeDisable

Disables attributes of a uDMA channel.

Prototype:
```
void
uDMAChannelAttributeDisable(unsigned long ulChannel,
                            unsigned long ulAttr)
```

Parameters:
ulChannel is the channel to configure.
ulAttr is a combination of attributes for the channel.

Description:
This function is used to disable attributes of a uDMA channel.

The *ulChannel* parameter must be one of the following:

- **UDMA_CHANNEL_UART0RX** for UART 0 receive channel
- **UDMA_CHANNEL_UART0TX** for UART 0 transmit channel
- **UDMA_CHANNEL_UART1RX** for UART 1 receive channel
- **UDMA_CHANNEL_UART1TX** for UART 1 transmit channel
- **UDMA_CHANNEL_SSI0RX** for SSI 0 receive channel
- **UDMA_CHANNEL_SSI0TX** for SSI 0 transmit channel
- **UDMA_CHANNEL_SSI1RX** for SSI 1 receive channel
- **UDMA_CHANNEL_SSI1TX** for SSI 1 transmit channel
- **UDMA_CHANNEL_SW** for the software dedicated uDMA channel

And for microcontrollers that have a USB peripheral:

- **UDMA_CHANNEL_USBEP1RX** for USB endpoint 1 receive
- **UDMA_CHANNEL_USBEP1TX** for USB endpoint 1 transmit
- **UDMA_CHANNEL_USBEP2RX** for USB endpoint 2 receive
- **UDMA_CHANNEL_USBEP2TX** for USB endpoint 2 transmit
- **UDMA_CHANNEL_USBEP3RX** for USB endpoint 3 receive

- **UDMA_CHANNEL_USBEP3TX** for USB endpoint 3 transmit

The *ulAttr* parameter is the logical OR of any of the following:

- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only a burst mode.
- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel.
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.
- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

Returns:
None.

23.2.2.2 uDMAChannelAttributeEnable

Enables attributes of a uDMA channel.

Prototype:
```
void
uDMAChannelAttributeEnable(unsigned long ulChannel,
                           unsigned long ulAttr)
```

Parameters:
ulChannel is the channel to configure.
ulAttr is a combination of attributes for the channel.

Description:
The *ulChannel* parameter must be one of the following:

- **UDMA_CHANNEL_UART0RX** for UART 0 receive channel
- **UDMA_CHANNEL_UART0TX** for UART 0 transmit channel
- **UDMA_CHANNEL_UART1RX** for UART 1 receive channel
- **UDMA_CHANNEL_UART1TX** for UART 1 transmit channel
- **UDMA_CHANNEL_SSI0RX** for SSI 0 receive channel
- **UDMA_CHANNEL_SSI0TX** for SSI 0 transmit channel
- **UDMA_CHANNEL_SSI1RX** for SSI 1 receive channel
- **UDMA_CHANNEL_SSI1TX** for SSI 1 transmit channel
- **UDMA_CHANNEL_SW** for the software dedicated uDMA channel

And for microcontrollers that have a USB peripheral:

- **UDMA_CHANNEL_USBEP1RX** for USB endpoint 1 receive
- **UDMA_CHANNEL_USBEP1TX** for USB endpoint 1 transmit
- **UDMA_CHANNEL_USBEP2RX** for USB endpoint 2 receive
- **UDMA_CHANNEL_USBEP2TX** for USB endpoint 2 transmit
- **UDMA_CHANNEL_USBEP3RX** for USB endpoint 3 receive
- **UDMA_CHANNEL_USBEP3TX** for USB endpoint 3 transmit

The *ulAttr* parameter is the logical OR of any of the following:

- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only a burst mode.

- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel.
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.
- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

Returns:
None.

23.2.2.3 uDMAChannelAttributeGet

Gets the enabled attributes of a uDMA channel.

Prototype:
```
unsigned long
uDMAChannelAttributeGet(unsigned long ulChannel)
```

Parameters:
ulChannel is the channel to configure.

Description:
This function returns a combination of flags representing the attributes of the uDMA channel.

The *ulChannel* parameter must be one of the following:

- **UDMA_CHANNEL_UART0RX** for UART 0 receive channel
- **UDMA_CHANNEL_UART0TX** for UART 0 transmit channel
- **UDMA_CHANNEL_UART1RX** for UART 1 receive channel
- **UDMA_CHANNEL_UART1TX** for UART 1 transmit channel
- **UDMA_CHANNEL_SSI0RX** for SSI 0 receive channel
- **UDMA_CHANNEL_SSI0TX** for SSI 0 transmit channel
- **UDMA_CHANNEL_SSI1RX** for SSI 1 receive channel
- **UDMA_CHANNEL_SSI1TX** for SSI 1 transmit channel
- **UDMA_CHANNEL_SW** for the software dedicated uDMA channel

And for microcontrollers that have a USB peripheral:

- **UDMA_CHANNEL_USBEP1RX** for USB endpoint 1 receive
- **UDMA_CHANNEL_USBEP1TX** for USB endpoint 1 transmit
- **UDMA_CHANNEL_USBEP2RX** for USB endpoint 2 receive
- **UDMA_CHANNEL_USBEP2TX** for USB endpoint 2 transmit
- **UDMA_CHANNEL_USBEP3RX** for USB endpoint 3 receive
- **UDMA_CHANNEL_USBEP3TX** for USB endpoint 3 transmit

Returns:
Returns the logical OR of the attributes of the uDMA channel, which can be any of the following:

- **UDMA_ATTR_USEBURST** is used to restrict transfers to use only a burst mode.
- **UDMA_ATTR_ALTSELECT** is used to select the alternate control structure for this channel.
- **UDMA_ATTR_HIGH_PRIORITY** is used to set this channel to high priority.
- **UDMA_ATTR_REQMASK** is used to mask the hardware request signal from the peripheral for this channel.

23.2.2.4 uDMAChannelControlSet

Sets the control parameters for a uDMA channel.

Prototype:

```
void
uDMAChannelControlSet(unsigned long ulChannel,
                      unsigned long ulControl)
```

Parameters:

ulChannel is the logical OR of the uDMA channel number with **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

ulControl is logical OR of several control values to set the control parameters for the channel.

Description:

This function is used to set control parameters for a uDMA transfer. These are typically parameters that are not changed often.

The *ulChannel* parameter is one of the choices documented in the uDMAChannelEnable() function. It should be the logical OR of the channel with one of **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose whether the primary or alternate data structure is used.

The *ulControl* parameter is the logical OR of five values: the data size, the source address increment, the destination address increment, the arbitration size, and the use burst flag. The choices available for each of these values is described below.

Choose the data size from one of **UDMA_SIZE_8**, **UDMA_SIZE_16**, or **UDMA_SIZE_32** to select a data size of 8, 16, or 32 bits.

Choose the source address increment from one of **UDMA_SRC_INC_8**, **UDMA_SRC_INC_16**, **UDMA_SRC_INC_32**, or **UDMA_SRC_INC_NONE** to select an address increment of 8-bit bytes, 16-bit halfwords, 32-bit words, or to select non-incrementing.

Choose the destination address increment from one of **UDMA_DST_INC_8**, **UDMA_DST_INC_16**, **UDMA_DST_INC_32**, or **UDMA_DST_INC_NONE** to select an address increment of 8-bit bytes, 16-bit halfwords, 32-bit words, or to select non-incrementing.

The arbitration size determines how many items are transferred before the uDMA controller re-arbitrates for the bus. Choose the arbitration size from one of **UDMA_ARB_1**, **UDMA_ARB_2**, **UDMA_ARB_4**, **UDMA_ARB_8**, through **UDMA_ARB_1024** to select the arbitration size from 1 to 1024 items, in powers of 2.

The value **UDMA_NEXT_USEBURST** is used to force the channel to only respond to burst requests at the tail end of a scatter-gather transfer.

Note:

The address increment cannot be smaller than the data size.

Returns:

None.

23.2.2.5 uDMAChannelDisable

Disables a uDMA channel for operation.

Prototype:

```
void
uDMAChannelDisable(unsigned long ulChannel)
```

Parameters:

ulChannel is the channel number to disable.

Description:

This function disables a specific uDMA channel. Once disabled, a channel will not respond to uDMA transfer requests until re-enabled via uDMAChannelEnable().

The *ulChannel* parameter must be one of the following:

- **UDMA_CHANNEL_UART0RX** for UART 0 receive channel
- **UDMA_CHANNEL_UART0TX** for UART 0 transmit channel
- **UDMA_CHANNEL_UART1RX** for UART 1 receive channel
- **UDMA_CHANNEL_UART1TX** for UART 1 transmit channel
- **UDMA_CHANNEL_SSI0RX** for SSI 0 receive channel
- **UDMA_CHANNEL_SSI0TX** for SSI 0 transmit channel
- **UDMA_CHANNEL_SSI1RX** for SSI 1 receive channel
- **UDMA_CHANNEL_SSI1TX** for SSI 1 transmit channel
- **UDMA_CHANNEL_SW** for the software dedicated uDMA channel

And for microcontrollers that have a USB peripheral:

- **UDMA_CHANNEL_USBEP1RX** for USB endpoint 1 receive
- **UDMA_CHANNEL_USBEP1TX** for USB endpoint 1 transmit
- **UDMA_CHANNEL_USBEP2RX** for USB endpoint 2 receive
- **UDMA_CHANNEL_USBEP2TX** for USB endpoint 2 transmit
- **UDMA_CHANNEL_USBEP3RX** for USB endpoint 3 receive
- **UDMA_CHANNEL_USBEP3TX** for USB endpoint 3 transmit

Returns:

None.

23.2.2.6 uDMAChannelEnable

Enables a uDMA channel for operation.

Prototype:

```
void
uDMAChannelEnable(unsigned long ulChannel)
```

Parameters:

ulChannel is the channel number to enable.

Description:

This function enables a specific uDMA channel for use. This function must be used to enable a channel before it can be used to perform a uDMA transfer.

When a uDMA transfer is completed, the channel will be automatically disabled by the uDMA controller. Therefore, this function should be called prior to starting up any new transfer.

The *ulChannel* parameter must be one of the following:

- **UDMA_CHANNEL_UART0RX** for UART 0 receive channel
- **UDMA_CHANNEL_UART0TX** for UART 0 transmit channel
- **UDMA_CHANNEL_UART1RX** for UART 1 receive channel
- **UDMA_CHANNEL_UART1TX** for UART 1 transmit channel
- **UDMA_CHANNEL_SSI0RX** for SSI 0 receive channel
- **UDMA_CHANNEL_SSI0TX** for SSI 0 transmit channel
- **UDMA_CHANNEL_SSI1RX** for SSI 1 receive channel
- **UDMA_CHANNEL_SSI1TX** for SSI 1 transmit channel
- **UDMA_CHANNEL_SW** for the software dedicated uDMA channel

And for microcontrollers that have a USB peripheral:

- **UDMA_CHANNEL_USBEP1RX** for USB endpoint 1 receive
- **UDMA_CHANNEL_USBEP1TX** for USB endpoint 1 transmit
- **UDMA_CHANNEL_USBEP2RX** for USB endpoint 2 receive
- **UDMA_CHANNEL_USBEP2TX** for USB endpoint 2 transmit
- **UDMA_CHANNEL_USBEP3RX** for USB endpoint 3 receive
- **UDMA_CHANNEL_USBEP3TX** for USB endpoint 3 transmit

Returns:
None.

23.2.2.7 uDMAChannelIsEnabled

Checks if a uDMA channel is enabled for operation.

Prototype:
```
tBoolean
uDMAChannelIsEnabled(unsigned long ulChannel)
```

Parameters:
ulChannel is the channel number to check.

Description:
This function checks to see if a specific uDMA channel is enabled. This can be used to check the status of a transfer, since the channel will be automatically disabled at the end of a transfer.

The *ulChannel* parameter must be one of the following:

- **UDMA_CHANNEL_UART0RX** for UART 0 receive channel
- **UDMA_CHANNEL_UART0TX** for UART 0 transmit channel
- **UDMA_CHANNEL_UART1RX** for UART 1 receive channel
- **UDMA_CHANNEL_UART1TX** for UART 1 transmit channel
- **UDMA_CHANNEL_SSI0RX** for SSI 0 receive channel
- **UDMA_CHANNEL_SSI0TX** for SSI 0 transmit channel
- **UDMA_CHANNEL_SSI1RX** for SSI 1 receive channel
- **UDMA_CHANNEL_SSI1TX** for SSI 1 transmit channel
- **UDMA_CHANNEL_SW** for the software dedicated uDMA channel

And for microcontrollers that have a USB peripheral:

- **UDMA_CHANNEL_USBEP1RX** for USB endpoint 1 receive
- **UDMA_CHANNEL_USBEP1TX** for USB endpoint 1 transmit
- **UDMA_CHANNEL_USBEP2RX** for USB endpoint 2 receive
- **UDMA_CHANNEL_USBEP2TX** for USB endpoint 2 transmit
- **UDMA_CHANNEL_USBEP3RX** for USB endpoint 3 receive
- **UDMA_CHANNEL_USBEP3TX** for USB endpoint 3 transmit

Returns:

Returns **true** if the channel is enabled, **false** if disabled.

23.2.2.8 uDMAChannelModeGet

Gets the transfer mode for a uDMA channel.

Prototype:

```
unsigned long
uDMAChannelModeGet(unsigned long ulChannel)
```

Parameters:

ulChannel is the logical or of the uDMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

Description:

This function is used to get the transfer mode for the uDMA channel. It can be used to query the status of a transfer on a channel. When the transfer is complete the mode will be **UDMA_MODE_STOP**.

The *ulChannel* parameter is one of the choices documented in the uDMAChannelEnable() function. It should be the logical OR of the channel with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose whether the primary or alternate data structure is used.

Returns:

Returns the transfer mode of the specified channel and control structure, which will be one of the following values: **UDMA_MODE_STOP**, **UDMA_MODE_BASIC**, **UDMA_MODE_AUTO**, **UDMA_MODE_PINGPONG**, **UDMA_MODE_MEM_SCATTER_GATHER**, or **UDMA_MODE_PER_SCATTER_GATHER**.

23.2.2.9 uDMAChannelRequest

Requests a uDMA channel to start a transfer.

Prototype:

```
void
uDMAChannelRequest(unsigned long ulChannel)
```

Parameters:

ulChannel is the channel number on which to request a uDMA transfer.

Description:

This function allows software to request a uDMA channel to begin a transfer. This could be used for performing a memory to memory transfer, or if for some reason a transfer needs to be initiated by software instead of the peripheral associated with that channel.

The *ulChannel* parameter must be one of the following:

- **UDMA_CHANNEL_UART0RX** for UART 0 receive channel
- **UDMA_CHANNEL_UART0TX** for UART 0 transmit channel
- **UDMA_CHANNEL_UART1RX** for UART 1 receive channel
- **UDMA_CHANNEL_UART1TX** for UART 1 transmit channel
- **UDMA_CHANNEL_SSI0RX** for SSI 0 receive channel
- **UDMA_CHANNEL_SSI0TX** for SSI 0 transmit channel
- **UDMA_CHANNEL_SSI1RX** for SSI 1 receive channel
- **UDMA_CHANNEL_SSI1TX** for SSI 1 transmit channel
- **UDMA_CHANNEL_SW** for the software dedicated uDMA channel

And for microcontrollers that have a USB peripheral:

- **UDMA_CHANNEL_USBEP1RX** for USB endpoint 1 receive
- **UDMA_CHANNEL_USBEP1TX** for USB endpoint 1 transmit
- **UDMA_CHANNEL_USBEP2RX** for USB endpoint 2 receive
- **UDMA_CHANNEL_USBEP2TX** for USB endpoint 2 transmit
- **UDMA_CHANNEL_USBEP3RX** for USB endpoint 3 receive
- **UDMA_CHANNEL_USBEP3TX** for USB endpoint 3 transmit

Note:

If the channel is **UDMA_CHANNEL_SW** and interrupts are used, then the completion will be signalled on the uDMA dedicated interrupt. If a peripheral channel is used, then the completion will be signalled on the peripheral's interrupt.

Returns:

None.

23.2.2.10 uDMAChannelSizeGet

Gets the current transfer size for a uDMA channel.

Prototype:

```
unsigned long
uDMAChannelSizeGet(unsigned long ulChannel)
```

Parameters:

ulChannel is the logical or of the uDMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

Description:

This function is used to get the uDMA transfer size for a channel. The transfer size is the number of items to transfer, where the size of an item might be 8, 16, or 32 bits. If a partial transfer has already occurred, then the number of remaining items will be returned. If the transfer is complete, then 0 will be returned.

The *ulChannel* parameter is one of the choices documented in the uDMAChannelEnable() function. It should be the logical OR of the channel with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose whether the primary or alternate data structure is used.

Returns:

Returns the number of items remaining to transfer.

23.2.2.11 uDMAChannelTransferSet

Sets the transfer parameters for a uDMA channel.

Prototype:

```
void
uDMAChannelTransferSet(unsigned long ulChannel,
                       unsigned long ulMode,
                       void *pvSrcAddr,
                       void *pvDstAddr,
                       unsigned long ulTransferSize)
```

Parameters:

ulChannel is the logical or of the uDMA channel number with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT**.

ulMode is the type of uDMA transfer.

pvSrcAddr is the source address for the transfer.

pvDstAddr is the destination address for the transfer.

ulTransferSize is the number of data items to transfer.

Description:

This function is used to set the parameters for a uDMA transfer. These are typically parameters that are changed often. The function uDMAChannelControlSet() MUST be called at least once for this channel prior to calling this function.

The *ulChannel* parameter is one of the choices documented in the uDMAChannelEnable() function. It should be the logical OR of the channel with either **UDMA_PRI_SELECT** or **UDMA_ALT_SELECT** to choose whether the primary or alternate data structure is used.

The *ulMode* parameter should be one of the following values:

- **UDMA_MODE_STOP** stops the uDMA transfer. The controller sets the mode to this value at the end of a transfer.
- **UDMA_MODE_BASIC** to perform a basic transfer based on request.
- **UDMA_MODE_AUTO** to perform a transfer that will always complete once started even if request is removed.
- **UDMA_MODE_PINGPONG** to set up a transfer that switches between the primary and alternate control structures for the channel. This allows use of ping-pong buffering for uDMA transfers.
- **UDMA_MODE_MEM_SCATTER_GATHER** to set up a memory scatter-gather transfer.
- **UDMA_MODE_PER_SCATTER_GATHER** to set up a peripheral scatter-gather transfer.

The *pvSrcAddr* and *pvDstAddr* parameters are pointers to the first location of the data to be transferred. These addresses should be aligned according to the item size. The compiler will take care of this if the pointers are pointing to storage of the appropriate data type.

The *ulTransferSize* parameter is the number of data items, not the number of bytes.

The two scatter/gather modes, memory and peripheral, are actually different depending on whether the primary or alternate control structure is selected. This function will look for the **UDMA_PRI_SELECT** and **UDMA_ALT_SELECT** flag along with the channel number and will set the scatter/gather mode as appropriate for the primary or alternate control structure.

The channel must also be enabled using uDMAChannelEnable() after calling this function. The transfer will not begin until the channel has been set up and enabled. Note that the channel is automatically disabled after the transfer is completed, meaning that uDMAChannelEnable() must be called again after setting up the next transfer.

Note:
Great care must be taken to not modify a channel control stucture that is in use or else the results will be unpredictable, including the possibility of undesired data transfers to or from memory or peripherals. For BASIC and AUTO modes, it is safe to make changes when the channel is disabled, or the uDMAChannelModeGet() returns **UDMA_MODE_STOP**. For PING-PONG or one of the SCATTER_GATHER modes, it is safe to modify the primary or alternate control structure only when the other is being used. The uDMAChannelModeGet() function will return **UDMA_MODE_STOP** when a channel control structure is inactive and safe to modify.

Returns:
None.

23.2.2.12 uDMAControlBaseGet

Gets the base address for the channel control table.

Prototype:
```
void *
uDMAControlBaseGet(void)
```

Description:
This function gets the base address of the channel control table. This table resides in system memory and holds control information for each uDMA channel.

Returns:
Returns a pointer to the base address of the channel control table.

23.2.2.13 uDMAControlBaseSet

Sets the base address for the channel control table.

Prototype:
```
void
uDMAControlBaseSet(void *pControlTable)
```

Parameters:
pControlTable is a pointer to the 1024 byte aligned base address of the uDMA channel control table.

Description:

This function sets the base address of the channel control table. This table resides in system memory and holds control information for each uDMA channel. The table must be aligned on a 1024 byte boundary. The base address must be set before any of the channel functions can be used.

The size of the channel control table depends on the number of uDMA channels, and which transfer modes are used. Refer to the introductory text and the microcontroller datasheet for more information about the channel control table.

Returns:

None.

23.2.2.14 uDMADisable

Disables the uDMA controller for use.

Prototype:

```
void
uDMADisable(void)
```

Description:

This function disables the uDMA controller. Once disabled, the uDMA controller will not operate until re-enabled with uDMAEnable().

Returns:

None.

23.2.2.15 uDMAEnable

Enables the uDMA controller for use.

Prototype:

```
void
uDMAEnable(void)
```

Description:

This function enables the uDMA controller. The uDMA controller must be enabled before it can be configured and used.

Returns:

None.

23.2.2.16 uDMAErrorStatusClear

Clears the uDMA error interrupt.

Prototype:

```
void
uDMAErrorStatusClear(void)
```

Description:
This function clears a pending uDMA error interrupt. It should be called from within the uDMA error interrupt handler to clear the interrupt.

Returns:
None.

23.2.2.17 uDMAErrorStatusGet

Gets the uDMA error status.

Prototype:
```
unsigned long
uDMAErrorStatusGet(void)
```

Description:
This function returns the uDMA error status. It should be called from within the uDMA error interrupt handler to determine if a uDMA error occurred.

Returns:
Returns non-zero if a uDMA error is pending.

23.2.2.18 uDMAIntRegister

Registers an interrupt handler for the uDMA controller.

Prototype:
```
void
uDMAIntRegister(unsigned long ulIntChannel,
                void (*pfnHandler)(void))
```

Parameters:
ulIntChannel identifies which uDMA interrupt is to be registered.
pfnHandler is a pointer to the function to be called when the interrupt is activated.

Description:
This sets and enables the handler to be called when the uDMA controller generates an interrupt. The *ulIntChannel* parameter should be one of the following:

- **UDMA_INT_SW** to register an interrupt handler to process interrupts from the uDMA software channel (UDMA_CHANNEL_SW)
- **UDMA_INT_ERR** to register an interrupt handler to process uDMA error interrupts

See also:
IntRegister() for important information about registering interrupt handlers.

Note:
The interrupt handler for uDMA is for transfer completion when the channel UDMA_CHANNEL_SW is used, and for error interrupts. The interrupts for each peripheral channel are handled through the individual peripheral interrupt handlers.

Returns:
None.

23.2.2.19 uDMAIntUnregister

Unregisters an interrupt handler for the uDMA controller.

Prototype:

```
void
uDMAIntUnregister(unsigned long ulIntChannel)
```

Parameters:

ulIntChannel identifies which uDMA interrupt to unregister.

Description:

This function will disable and clear the handler to be called for the specified uDMA interrupt. The *ulIntChannel* parameter should be one of **UDMA_INT_SW** or **UDMA_INT_ERR** as documented for the function uDMAIntRegister().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

23.3 Programming Example

The following example sets up the uDMA controller to perform a software initiated memory-to-memory transfer:

```
//
// The application must allocate the channel control table.
// This one is a full table for all modes and channels.
// NOTE: This table must be 1024 byte aligned.
//
unsigned char ucDMAControlTable[1024];

//
// Source and destination buffers used for the DMA transfer.
//
unsigned char ucSourceBuffer[256];
unsigned char ucDestBuffer[256];

//
// Enable the uDMA controller.
//
uDMAEnable();

//
// Set the base for the channel control table.
//
uDMAControlBaseSet(&ucDMAControlTable[0]);

//
// No attributes need to be set for a software based transfer.
// They will be cleared by default, but are explicitly cleared
// here, in case they were set elsewhere.
//
uDMAChannelAttributeDisable(UDMA_CONFIG_ALL);
```

```
//
// Now set up the characteristics of the transfer.  It will
// be 8 bit data size, with source and destination increments
// in bytes, to perform a byte-wise buffer copy.  A bus arbitration
// size of 8 is used.
//
uDMAChannelControlSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                      UDMA_SIZE_8 | UDMA_SRC_INC_8 |
                      UDMA_DST_INC_8 | UDMA_ARB_8);

//
// The transfer buffers and transfer size will now  be configured.
// The transfer will use AUTO mode, which means that the
// transfer will automatically run to completion after the first
// request.
//
uDMAChannelTransferSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                       UDMA_MODE_AUTO, ucSourceBuffer, ucDestBuffer,
                       sizeof(ucDestBuffer));

//
// Finally, the channel must be enabled.  Since this is a software
// initiated transfer, a request must also be made.  This will
// start the transfer running.
//
uDMAChannelEnable(UDMA_CHANNEL_SW);
uDMAChannelRequest(UDMA_CHANNEL_SW);
```

24 USB Controller

24.1 Introduction

The USB APIs provide a set of functions that are used to access the Stellaris USB device or host controllers. The APIs are split into groups according to the functionality provided by the USB controller present in the microcontroller. Because of this, the driver has to handle microcontrollers that have only a USB device interface, a host and/or device interface, or microcontrollers that have an OTG interface, The groups are the following: USBDev, USBHost, USBOTG, USBEndpoint, and USBFIFO. The APIs in the USBDev group are only used with microcontrollers that have a USB device controller. The APIs in the USBHost group can only be used with microcontrollers that have a USB host controller. The USBOTG APIs are used by microcontrollers with an OTG interface. With USB OTG controllers, once the mode of the USB controller is configured, the device or host APIs should be used. The remainder of the APIs are used for both USB host and USB device controllers. The USBEndpoint APIs are used to configure and access the endpoints while the USBFIFO APIs are used to configure the size and location of the FIFOs.

24.2 Using USB with the uDMA Controller

The USB controller can be used with the uDMA for either sending or receiving data with both host and device controllers. The uDMA controller cannot be used to access endpoint 0, however all other endpoints are capable of using the uDMA controller. The uDMA channel numbers for USB are defined by the following values:

- DMA_CHANNEL_USBEP1RX
- DMA_CHANNEL_USBEP1TX
- DMA_CHANNEL_USBEP2RX
- DMA_CHANNEL_USBEP2TX
- DMA_CHANNEL_USBEP3RX
- DMA_CHANNEL_USBEP3TX

Since the uDMA controller views transfers as either transmit or receive, and the USB controller operates on IN/OUT transactions, some care must be taken to use the correct uDMA channel with the correct endpoint. USB host IN and USB device OUT endpoints both use receive uDMA channels while USB host OUT and USB device IN endpoints will use transmit uDMA channels.

When configuring the endpoint there are additional DMA settings needed. When calling USBDevEndpointConfig() for an endpoint that will use uDMA, extra flags need to be added to the *ulFlags* parameter. These flags are one of **USB_EP_DMA_MODE_0** or **USB_EP_DMA_MODE_1** to control the mode of the DMA transaction, and likely **USB_EP_AUTO_SET** to allow the data to be

transmitted automatically once a packet is ready. **USB_EP_DMA_MODE_0** will generate an interrupt whenever there is more space available in the FIFO. This allows the application code to perform operations between each packet.**USB_EP_DMA_MODE_1** will only interrupt when the DMA transfer complete or there is some type of error condition. This can be used for larger transmissions that require no interaction between packets. **USB_EP_AUTO_SET** should normally be specified when using uDMA to prevent the need for application code to start the actual transfer of data.

Example: Endpoint configuration for a device IN endpoint:

```
//
// Endpoint 1 is a device mode BULK IN endpoint using DMA.
//
USBDevEndpointConfig(
    USB0_BASE,
    USB_EP_1,
    64,
    USB_EP_MODE_BULK | USB_EP_DEV_IN |
    USB_EP_DMA_MODE_0 | USB_EP_AUTO_SET);
```

The application must provide the configuration of the actual uDMA controller. First, to clear out any previous settings, the application should call DMAChannelAttributeClear(). Then the application should call DMAChannelAttributeSet() for the uDMA channel that corresponds to the endpoint, and specify the **DMA_CONFIG_USEBURST** flag.

Note:
All uDMA transfers used by the USB controller must enable burst mode.

The application needs to indicate the size of each DMA transactions, combined with the source and destination increments and the arbitration level for the uDMA controller.

Example: Configure endpoint 1 transmit channel.

```
//
// Set up the DMA for USB transmit.
//
DMAChannelAttributeClear(
    DMA_CHANNEL_USBEP1TX,
    DMA_CONFIG_ALL);

//
// Enable uDMA burst mode.
//
DMAChannelAttributeSet(
    DMA_CHANNEL_USBEP1TX,
    DMA_CONFIG_USEBURST);

//
// Data size is 8 bits and the source has a one byte increment.
// Destination has no increment as it is a FIFO.
//
DMAChannelControlSet(
    DMA_CHANNEL_USBEP1TX,
    DMA_DATA_SIZE_8,
    DMA_ADDR_INC_8,
    DMA_ADDR_INC_NONE,
    DMA_ARB_64,
    0);
```

The next step is to actually start the uDMA transfer once the data is ready to be sent. There are the only two calls that the application needs to call to start a new transfer. Normally all of the previous

uDMA configuration can stay the same. The first call, DMAChannelTransferSet(), resets the source and destination addresses for the DMA transfer and specifies how much data will be sent. The next call, DMAChannelEnable() actually allows the DMA controller to begin requesting data.

Example: Start the transfer of data on endpoint 1.

```
//
// Configure the address and size of the data to transfer.
//
DMAChannelTransferSet(
    DMA_CHANNEL_USBEP1TX,
    DMA_MODE_BASIC,
    pData,
    USBFIFOAddr(USB0_BASE, USB_EP_1),
    64);
//
// Start the transfer.
//
DMAChannelEnable(DMA_CHANNEL_USBEP1TX);
```

Because the uDMA interrupt occurs on the same interrupt vector as any other USB interrupt, the application must perform an extra check to determine what was the actual source of the interrupt. It is important to note that this DMA interrupt does not mean that the USB transfer is complete, but that the data has been transferred to the USB controller's FIFO. There will also be an interrupt indicating that the USB transfer is complete. However, both events need to be handled in the same interrupt routine. This because if other code in the system holds off the USB interrupt routine, both the uDMA complete and transfer complete can occur before the USB interrupt handler is called. The USB has no status bit indicating that the interrupt was due to a DMA complete, which means that the application must remember if a DMA transaction was in progress. The example below shows the g_ulFlags global variable being used to remember that a DMA transfer was pending.

Example: Interrupt handling with uDMA.

```
if((g_ulFlags & EP1_DMA_IN_PEND) &&
   (DMAChannelModeGet(DMA_CHANNEL_USBEP1TX) == DMA_MODE_STOP))
{
    //
    // Handle the DMA complete case.
    //
    ...
}

//
// Get the interrupt status.
//
ulStatus = USBIntStatus(USB0_BASE);

if(ulStatus & USB_INT_DEV_IN_EP1)
{
    //
    // Handler the transfer complete case.
    //
    ...
}
```

To use the USB device controller with an OUT endpoint, the application must use a receive uDMA channel. When calling USBDevEndpointConfig() for an endpoint that uses uDMA, the application must set extra flags in the *ulFlags* parameter. The **USB_EP_DMA_MODE_0** and **USB_EP_DMA_MODE_1** control the mode of the transaction, **USB_EP_AUTO_CLEAR** allows the data to be received automatically without needing to manually acknowledge that the data has been

read. **USB_EP_DMA_MODE_0** will not generate an interrupt when each packet is sent over USB and will only interrupt when the DMA transfer is complete. **USB_EP_DMA_MODE_1** will interrupt when the DMA transfer complete or a short packet is received. This is useful for BULK endpoints that may not have prior knowledge of how much data is being received. **USB_EP_AUTO_CLEAR** should normally be specified when using uDMA to prevent the need for application code to acknowledge that the data has been read from the FIFO. The example below configures endpoint 1 as a Device mode Bulk OUT endpoint using DMA mode 1 with a max packet size of 64 bytes.

Example: Configure endpoint 1 receive channel:

```
//
// Endpoint 1 is a device mode BULK OUT endpoint using DMA.
//
USBDevEndpointConfig(
    USB0_BASE,
    USB_EP_1,
    64,
    USB_EP_DEV_OUT | USB_EP_MODE_BULK |
    USB_EP_DMA_MODE_1 | USB_EP_AUTO_CLEAR);
```

Next the configuration of the actual uDMA controller is needed. Like the transmit case, the first a call to DMAChannelAttributeClear() is made to clear any previous settings. This is followed by a call to DMAChannelAttributeSet() with the **DMA_CONFIG_USEBURST** value.

Note:
All uDMA transfers used by the USB controller must use burst mode.

The final call sets the read access size to 8 bits wide, the source address increment to 0, the destination address increment to 8 bits and the uDMA arbitration size to 64 bytes.

Example: Configure endpoint 1 transmit channel.

```
//
// Clear out any uDMA settings.
//
DMAChannelAttributeClear(
    DMA_CHANNEL_USBEP1RX,
    DMA_CONFIG_ALL);

DMAChannelAttributeSet(
    DMA_CHANNEL_USBEP1RX,
    DMA_CONFIG_USEBURST);

DMAChannelControlSet(
    DMA_CHANNEL_USBEP1RX,
    DMA_DATA_SIZE_8,
    DMA_ADDR_INC_NONE,
    DMA_ADDR_INC_8,
    DMA_ARB_64,
    0);
```

The next step is to actually start the uDMA transfer. Unlike the transfer side, if the application is ready, this can be set up right away to wait for incoming data. Like the transmit case, these are the only calls needed to start a new transfer, normally all of the previous uDMA configuration can remain the same.

Example: Start requesting of data on endpoint 1.

```
//
```

```
// Configure the address and size of the data to transfer.  The transfer
// is from the USB FIFO for endpoint 0 to g_DataBufferIn.
//
DMAChannelTransferSet(
    DMA_CHANNEL_USBEP1RX,
    DMA_MODE_BASIC,
    USBFIFOAddr(USB0_BASE, USB_EP_1),
    g_DataBufferIn,
    64);

//
// Enable the uDMA channel and wait for data.
//
DMAChannelEnable(DMA_CHANNEL_USBEP1RX);
```

The uDMA interrupt occurs on the same interrupt vector as any other USB interrupt, this means that the application needs to check to see what was the actual source of the interrupt. It is possible that the USB interrupt does not indicate that the USB transfer was complete. The interrupt could also have been caused by a short packet, error, or even a transmit complete. This requires that the application check both receive cases to determine if this is related to receiving data on the endpoint. Because the USB has no status bit indicating that the interrupt was due to a DMA complete, the application must remember if a DMA transaction was in progress.

Example: Interrupt handling with uDMA.

```
//
// Get the current interrupt status.
//
ulStatus = USBIntStatus(USB0_BASE);

if(ulStatus & USB_INT_DEV_OUT_EP1)
{
    //
    // Handle a short packet.
    //
    ...
}
else if((g_ulFlags & EP1_DMA_OUT_PEND) &&
        (DMAChannelModeGet(DMA_CHANNEL_USBEP1RX) == DMA_MODE_STOP)
{
    //
    // Handle the DMA complete case.
    //
    ...

    //
    // Restart receive DMA if desired.
    //
    ...
}
```

24.3 API Functions

Functions

- unsigned long USBDevAddrGet (unsigned long ulBase)
- void USBDevAddrSet (unsigned long ulBase, unsigned long ulAddress)
- void USBDevConnect (unsigned long ulBase)

- void USBDevDisconnect (unsigned long ulBase)
- void USBDevEndpointConfig (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulMaxPacketSize, unsigned long ulFlags)
- void USBDevEndpointConfigGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long *pulMaxPacketSize, unsigned long *pulFlags)
- void USBDevEndpointDataAck (unsigned long ulBase, unsigned long ulEndpoint, tBoolean bIsLastPacket)
- void USBDevEndpointStall (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- void USBDevEndpointStallClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- void USBDevEndpointStatusClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- unsigned long USBEndpointDataAvail (unsigned long ulBase, unsigned long ulEndpoint)
- long USBEndpointDataGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned char *pucData, unsigned long *pulSize)
- long USBEndpointDataPut (unsigned long ulBase, unsigned long ulEndpoint, unsigned char *pucData, unsigned long ulSize)
- long USBEndpointDataSend (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulTransType)
- void USBEndpointDataToggleClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- unsigned long USBEndpointStatus (unsigned long ulBase, unsigned long ulEndpoint)
- unsigned long USBFIFOAddrGet (unsigned long ulBase, unsigned long ulEndpoint)
- void USBFIFOConfigGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long *pulFIFOAddress, unsigned long *pulFIFOSize, unsigned long ulFlags)
- void USBFIFOConfigSet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFIFOAddress, unsigned long ulFIFOSize, unsigned long ulFlags)
- void USBFIFOFlush (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- unsigned long USBFrameNumberGet (unsigned long ulBase)
- unsigned long USBHostAddrGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- void USBHostAddrSet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulAddr, unsigned long ulFlags)
- void USBHostEndpointConfig (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulMaxPayload, unsigned long ulNAKPollInterval, unsigned long ulTargetEndpoint, unsigned long ulFlags)
- void USBHostEndpointDataAck (unsigned long ulBase, unsigned long ulEndpoint)
- void USBHostEndpointDataToggle (unsigned long ulBase, unsigned long ulEndpoint, tBoolean bDataToggle, unsigned long ulFlags)
- void USBHostEndpointStatusClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- unsigned long USBHostHubAddrGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)
- void USBHostHubAddrSet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulAddr, unsigned long ulFlags)
- void USBHostPwrDisable (unsigned long ulBase)
- void USBHostPwrEnable (unsigned long ulBase)

- void USBHostPwrFaultConfig (unsigned long ulBase, unsigned long ulFlags)
- void USBHostPwrFaultDisable (unsigned long ulBase)
- void USBHostPwrFaultEnable (unsigned long ulBase)
- void USBHostRequestIN (unsigned long ulBase, unsigned long ulEndpoint)
- void USBHostRequestStatus (unsigned long ulBase)
- void USBHostReset (unsigned long ulBase, tBoolean bStart)
- void USBHostResume (unsigned long ulBase, tBoolean bStart)
- unsigned long USBHostSpeedGet (unsigned long ulBase)
- void USBHostSuspend (unsigned long ulBase)
- void USBIntDisable (unsigned long ulBase, unsigned long ulFlags)
- void USBIntEnable (unsigned long ulBase, unsigned long ulFlags)
- void USBIntRegister (unsigned long ulBase, void (*pfnHandler)(void))
- unsigned long USBIntStatus (unsigned long ulBase)
- void USBIntUnregister (unsigned long ulBase)
- void USBOTGSessionRequest (unsigned long ulBase, tBoolean bStart)

24.3.1 Detailed Description

The USB APIs provide all of the functions needed by an application to implement a USB device or USB host stack. The APIs abstract the IN/OUT nature of endpoints based on the type of USB controller that is in use. Any API that uses the IN/OUT terminology will comply with the standard USB interpretation of these terms. For example, an OUT endpoint on a microcontroller that has only a device interface will actually receive data on this endpoint, while a microcontroller that has a host interface will actually transmit data on an OUT endpoint.

Another important fact to understand is that all endpoints in the USB controller, whether host or device, have two "sides" to them. This allows each endpoint to both transmit and receive data. An application can use a single endpoint for both and IN and OUT transactions. For example: In device mode, endpoint 1 could be configured to have BULK IN and BULK OUT handled by endpoint 1. It is important to note that the endpoint number used is the endpoint number reported to the host. For microcontrollers with host controllers, the application can use an endpoint communicate with both IN and OUT endpoints of different types as well. For example: Endpoint 2 could be used to communicate with one device's interrupt IN endpoint and another device's bulk OUT endpoint at the same time. This effectively gives the application one dedicated control endpoint for IN or OUT control transactions on endpoint 0, and three IN endpoints and three OUT endpoints.

The USB controller has a configurable FIFOs in devices that have a USB device controller as well as those that have a host controller. The overall size of the FIFO RAM is 4096 bytes. It is important to note that the first 64 bytes of this memory are dedicated to endpoint 0 for control transactions. The remaining 4032 bytes are configurable however the application desires. The FIFO configuration is usually set at the beginning of the application and not modified once the USB controller is in use. The FIFO configuration uses the USBFIFOConfig() API to set the starting address and the size of the FIFOs that are dedicated to each endpoint.

Example: FIFO Configuration

0-64 - endpoint 0 IN/OUT (64 bytes).

64-576 - endpoint 1 IN (512 bytes).

576-1088 - endpoint 1 OUT (512 bytes).

1088-1600 - endpoint 2 IN (512 bytes).

```
//
// FIFO for endpoint 1 IN starts at address 64 and is 512 bytes in size.
//
USBFIFOConfig(USB0_BASE, USB_EP_1, 64, USB_FIFO_SZ_512, USB_EP_DEV_IN);

//
// FIFO for endpoint 1 OUT starts at address 576 and is 512 bytes in size.
//
USBFIFOConfig(USB0_BASE, USB_EP_1, 576,
              USB_FIFO_SZ_512, USB_EP_DEV_OUT);

//
// FIFO for endpoint 2 IN starts at address 1088 and is 512 bytes in size.
//
USBFIFOConfig(USB0_BASE, USB_EP_2, 1088, USB_FIFO_SZ_512, USB_EP_DEV_IN);
```

24.3.2 Function Documentation

24.3.2.1 USBDevAddrGet

Returns the current device address in device mode.

Prototype:

```
unsigned long
USBDevAddrGet(unsigned long ulBase)
```

Parameters:

ulBase specifies the USB module base address.

Description:

This function will return the current device address. This address was set by a call to USBDevAddrSet().

Note:

This function should only be called in device mode.

Returns:

The current device address.

24.3.2.2 USBDevAddrSet

Sets the address in device mode.

Prototype:

```
void
USBDevAddrSet(unsigned long ulBase,
              unsigned long ulAddress)
```

Parameters:

ulBase specifies the USB module base address.

ulAddress is the address to use for a device.

Description:

This function will set the device address on the USB bus. This address was likely received via a SET ADDRESS command from the host controller.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.2.3 USBDevConnect

Connects the USB controller to the bus in device mode.

Prototype:
```
void
USBDevConnect(unsigned long ulBase)
```

Parameters:
ulBase specifies the USB module base address.

Description:
This function will cause the soft connect feature of the USB controller to be enabled. Call USBDisconnect() to remove the USB device from the bus.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.2.4 USBDevDisconnect

Removes the USB controller from the bus in device mode.

Prototype:
```
void
USBDevDisconnect(unsigned long ulBase)
```

Parameters:
ulBase specifies the USB module base address.

Description:
This function will cause the soft connect feature of the USB controller to remove the device from the USB bus. A call to USBDevConnect() is needed to reconnect to the bus.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.2.5 USBDevEndpointConfig

Sets the configuration for an endpoint.

Prototype:

```
void
USBDevEndpointConfig(unsigned long ulBase,
                     unsigned long ulEndpoint,
                     unsigned long ulMaxPacketSize,
                     unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulMaxPacketSize is the maximum packet size for this endpoint.

ulFlags are used to configure other endpoint settings.

Description:

This function will set the basic configuration for an endpoint in device mode. Endpiont zero does not have a dynamic configuration, so this function should not be called for endpoint zero. The *ulFlags* parameter determines some of the configuration while the other parameters provide the rest.

The **USB_EP_MODE_** flags define what the type is for the given endpoint.

- **USB_EP_MODE_CTRL** is a control endpoint.
- **USB_EP_MODE_ISOC** is an isochronous endpoint.
- **USB_EP_MODE_BULK** is a bulk endpoint.
- **USB_EP_MODE_INT** is an interrupt endpoint.

The **USB_EP_DMA_MODE_** flags determines the type of DMA access to the endpoint data FIFOs. The choice of the DMA mode depends on how the DMA controller is configured and how it is being used. See the “Using USB with the uDMA Controller” section for more information on DMA configuration.

When configuring an IN endpoint, the **USB_EP_AUTO_SET** bit can be specified to cause the automatic transmission of data on the USB bus as soon as *ulMaxPacketSize* bytes of data are written into the FIFO for this endpoint. This is commonly used with DMA as no interaction is required to start the transmission of data.

When configuring an OUT endpoint, the **USB_EP_AUTO_REQUEST** bit is specified to trigger the request for more data once the FIFO has been drained enough to receive *ulMaxPacketSize* more bytes of data. Also for OUT endpoints, the **USB_EP_AUTO_CLEAR** bit can be used to clear the data packet ready flag automatically once the data has been read from the FIFO. If this is not used, this flag must be manually cleared via a call to USBDevEndpointStatusClear(). Both of these settings can be used to remove the need for extra calls when using the controller in DMA mode.

Note:

This function should only be called in device mode.

Returns:

None.

24.3.2.6 USBDevEndpointConfigGet

Gets the current configuration for an endpoint.

Prototype:

```
void
USBDevEndpointConfigGet(unsigned long ulBase,
                        unsigned long ulEndpoint,
                        unsigned long *pulMaxPacketSize,
                        unsigned long *pulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

pulMaxPacketSize is a pointer which will be written with the maximum packet size for this endpoint.

pulFlags is a pointer which will be written with the current endpoint settings. On entry to the function, this pointer must contain either **USB_EP_DEV_IN** or **USB_EP_DEV_OUT** to indicate whether the IN or OUT endpoint is to be queried.

Description:

This function will return the basic configuration for an endpoint in device mode. The values returned in *∗pulMaxPacketSize* and *∗pulFlags* are equivalent to the *ulMaxPacketSize* and *ulFlags* previously passed to USBDevEndpointConfig for this endpoint.

Note:

This function should only be called in device mode.

Returns:

None.

24.3.2.7 USBDevEndpointDataAck

Acknowledge that data was read from the given endpoint's FIFO in device mode.

Prototype:

```
void
USBDevEndpointDataAck(unsigned long ulBase,
                      unsigned long ulEndpoint,
                      tBoolean bIsLastPacket)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

bIsLastPacket indicates if this is the last packet.

Description:

This function acknowledges that the data was read from the endpoint's FIFO. The *bIsLastPacket* parameter is set to a **true** value if this is the last in a series of data packets on endpoint zero. The *bIsLastPacket* parameter is not used for endpoints other than endpoint zero. This call can be used if processing is required between reading the data and acknowledging that the data has been read.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.2.8 USBDevEndpointStall

Stalls the specified endpoint in device mode.

Prototype:
```
void
USBDevEndpointStall(unsigned long ulBase,
                    unsigned long ulEndpoint,
                    unsigned long ulFlags)
```

Parameters:
ulBase specifies the USB module base address.
ulEndpoint specifies the endpoint to stall.
ulFlags specifies whether to stall the IN or OUT endpoint.

Description:
This function will cause to endpoint number passed in to go into a stall condition. If the *ulFlags* parameter is **USB_EP_DEV_IN** then the stall will be issued on the IN portion of this endpoint. If the *ulFlags* parameter is **USB_EP_DEV_OUT** then the stall will be issued on the OUT portion of this endpoint.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.2.9 USBDevEndpointStallClear

Clears the stall condition on the specified endpoint in device mode.

Prototype:
```
void
USBDevEndpointStallClear(unsigned long ulBase,
                         unsigned long ulEndpoint,
                         unsigned long ulFlags)
```

Parameters:
ulBase specifies the USB module base address.
ulEndpoint specifies which endpoint to remove the stall condition.
ulFlags specifies whether to remove the stall condition from the IN or the OUT portion of this endpoint.

Description:
This function will cause the endpoint number passed in to exit the stall condition. If the *ulFlags* parameter is **USB_EP_DEV_IN** then the stall will be cleared on the IN portion of this endpoint. If the *ulFlags* parameter is **USB_EP_DEV_OUT** then the stall will be cleared on the OUT portion of this endpoint.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.2.10 USBDevEndpointStatusClear

Clears the status bits in this endpoint in device mode.

Prototype:
```
void
USBDevEndpointStatusClear(unsigned long ulBase,
                          unsigned long ulEndpoint,
                          unsigned long ulFlags)
```

Parameters:
ulBase specifies the USB module base address.
ulEndpoint is the endpoint to access.
ulFlags are the status bits that will be cleared.

Description:
This function will clear the status of any bits that are passed in the *ulFlags* parameter. The *ulFlags* parameter can take the value returned from the USBEndpointStatus() call.

Note:
This function should only be called in device mode.

Returns:
None.

24.3.2.11 USBEndpointDataAvail

Determine the number of bytes of data available in a given endpoint's FIFO.

Prototype:
```
unsigned long
USBEndpointDataAvail(unsigned long ulBase,
                     unsigned long ulEndpoint)
```

Parameters:
ulBase specifies the USB module base address.
ulEndpoint is the endpoint to access.

Description:

This function will return the number of bytes of data currently available in the FIFO for the given receive (OUT) endpoint. It may be used prior to calling USBEndpointDataGet() to determine the size of buffer required to hold the newly-received packet.

Returns:

This call will return the number of bytes available in a given endpoint FIFO.

24.3.2.12 USBEndpointDataGet

Retrieves data from the given endpoint's FIFO.

Prototype:

```
long
USBEndpointDataGet(unsigned long ulBase,
                   unsigned long ulEndpoint,
                   unsigned char *pucData,
                   unsigned long *pulSize)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

pucData is a pointer to the data area used to return the data from the FIFO.

pulSize is initially the size of the buffer passed into this call via the *pucData* parameter. It will be set to the amount of data returned in the buffer.

Description:

This function will return the data from the FIFO for the given endpoint. The *pulSize* parameter should indicate the size of the buffer passed in the *pulData* parameter. The data in the *pulSize* parameter will be changed to match the amount of data returned in the *pucData* parameter. If a zero byte packet was received this call will not return a error but will instead just return a zero in the *pulSize* parameter. The only error case occurs when there is no data packet available.

Returns:

This call will return 0, or -1 if no packet was received.

24.3.2.13 USBEndpointDataPut

Puts data into the given endpoint's FIFO.

Prototype:

```
long
USBEndpointDataPut(unsigned long ulBase,
                   unsigned long ulEndpoint,
                   unsigned char *pucData,
                   unsigned long ulSize)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

pucData is a pointer to the data area used as the source for the data to put into the FIFO.
ulSize is the amount of data to put into the FIFO.

Description:
This function will put the data from the *pucData* parameter into the FIFO for this endpoint. If a packet is already pending for transmission then this call will not put any of the data into the FIFO and will return -1. Care should be taken to not write more data than can fit into the FIFO allocated by the call to USBFIFOConfig().

Returns:
This call will return 0 on success, or -1 to indicate that the FIFO is in use and cannot be written.

24.3.2.14 USBEndpointDataSend

Starts the transfer of data from an endpoint's FIFO.

Prototype:
```
long
USBEndpointDataSend(unsigned long ulBase,
                    unsigned long ulEndpoint,
                    unsigned long ulTransType)
```

Parameters:
ulBase specifies the USB module base address.
ulEndpoint is the endpoint to access.
ulTransType is set to indicate what type of data is being sent.

Description:
This function will start the transfer of data from the FIFO for a given endpoint. This is necessary if the **USB_EP_AUTO_SET** bit was not enabled for the endpoint. Setting the *ulTransType* parameter will allow the appropriate signaling on the USB bus for the type of transaction being requested. The *ulTransType* parameter should be one of the following:

- USB_TRANS_OUT for OUT transaction on any endpoint in host mode.
- USB_TRANS_IN for IN transaction on any endpoint in device mode.
- USB_TRANS_IN_LAST for the last IN transactions on endpoint zero in a sequence of IN transactions.
- USB_TRANS_SETUP for setup transactions on endpoint zero.
- USB_TRANS_STATUS for status results on endpoint zero.

Returns:
This call will return 0 on success, or -1 if a transmission is already in progress.

24.3.2.15 USBEndpointDataToggleClear

Sets the Data toggle on an endpoint to zero.

Prototype:
```
void
USBEndpointDataToggleClear(unsigned long ulBase,
```

```
                                    unsigned long ulEndpoint,
                                    unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint specifies the endpoint to reset the data toggle.

ulFlags specifies whether to access the IN or OUT endpoint.

Description:

This function will cause the controller to clear the data toggle for an endpoint. This call is not valid for endpoint zero and can be made with host or device controllers.

The *ulFlags* parameter should be one of **USB_EP_HOST_OUT**, **USB_EP_HOST_IN**, **USB_EP_DEV_OUT**, or **USB_EP_DEV_IN**.

Returns:

None.

24.3.2.16 USBEndpointStatus

Returns the current status of an endpoint.

Prototype:

```
unsigned long
USBEndpointStatus(unsigned long ulBase,
                  unsigned long ulEndpoint)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

Description:

This function will return the status of a given endpoint. If any of these status bits need to be cleared, then these these values must be cleared by calling the USBDevEndpointStatusClear() or USBHostEndpointStatusClear() functions.

The following are the status flags for host mode:

- **USB_HOST_IN_PID_ERROR** - PID error on the given endpoint.
- **USB_HOST_IN_NOT_COMP** - The device failed to respond to an IN request.
- **USB_HOST_IN_STALL** - A stall was received on an IN endpoint.
- **USB_HOST_IN_DATA_ERROR** - There was a CRC or bit-stuff error on an IN endpoint in Isochronous mode.
- **USB_HOST_IN_NAK_TO** - NAKs received on this IN endpoint for more than the specified timeout period.
- **USB_HOST_IN_ERROR** - Failed to communicate with a device using this IN endpoint.
- **USB_HOST_IN_FIFO_FULL** - This IN endpoint's FIFO is full.
- **USB_HOST_IN_PKTRDY** - Data packet ready on this IN endpoint.
- **USB_HOST_OUT_NAK_TO** - NAKs received on this OUT endpoint for more than the specified timeout period.
- **USB_HOST_OUT_NOT_COMP** - The device failed to respond to an OUT request.

- **USB_HOST_OUT_STALL** - A stall was received on this OUT endpoint.
- **USB_HOST_OUT_ERROR** - Failed to communicate with a device using this OUT endpoint.
- **USB_HOST_OUT_FIFO_NE** - This endpoint's OUT FIFO is not empty.
- **USB_HOST_OUT_PKTPEND** - The data transfer on this OUT endpoint has not completed.
- **USB_HOST_EP0_NAK_TO** - NAKs received on endpoint zero for more than the specified timeout period.
- **USB_HOST_EP0_ERROR** - The device failed to respond to a request on endpoint zero.
- **USB_HOST_EP0_IN_STALL** - A stall was received on endpoint zero for an IN transaction.
- **USB_HOST_EP0_IN_PKTRDY** - Data packet ready on endpoint zero for an IN transaction.

The following are the status flags for device mode:

- **USB_DEV_OUT_SENT_STALL** - A stall was sent on this OUT endpoint.
- **USB_DEV_OUT_DATA_ERROR** - There was a CRC or bit-stuff error on an OUT endpoint.
- **USB_DEV_OUT_OVERRUN** - An OUT packet was not loaded due to a full FIFO.
- **USB_DEV_OUT_FIFO_FULL** - The OUT endpoint's FIFO is full.
- **USB_DEV_OUT_PKTRDY** - There is a data packet ready in the OUT endpoint's FIFO.
- **USB_DEV_IN_NOT_COMP** - A larger packet was split up, more data to come.
- **USB_DEV_IN_SENT_STALL** - A stall was sent on this IN endpoint.
- **USB_DEV_IN_UNDERRUN** - Data was requested on the IN endpoint and no data was ready.
- **USB_DEV_IN_FIFO_NE** - The IN endpoint's FIFO is not empty.
- **USB_DEV_IN_PKTPEND** - The data transfer on this IN endpoint has not completed.
- **USB_DEV_EP0_SETUP_END** - A control transaction ended before Data End condition was sent.
- **USB_DEV_EP0_SENT_STALL** - A stall was sent on endpoint zero.
- **USB_DEV_EP0_IN_PKTPEND** - The data transfer on endpoint zero has not completed.
- **USB_DEV_EP0_OUT_PKTRDY** - There is a data packet ready in endpoint zero's OUT FIFO.

Returns:

The current status flags for the endpoint depending on mode.

24.3.2.17 USBFIFOAddrGet

Returns the absolute FIFO address for a given endpoint.

Prototype:

```
unsigned long
USBFIFOAddrGet(unsigned long ulBase,
               unsigned long ulEndpoint)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint specifies which endpoint's FIFO address to return.

Description:

This function returns the actual physical address of the FIFO. This is needed when the USB is going to be used with the uDMA controller and the source or destination address needs to be set to the physical FIFO address for a given endpoint.

Returns:

None.

24.3.2.18 USBFIFOConfigGet

Returns the FIFO configuration for an endpoint.

Prototype:

```
void
USBFIFOConfigGet(unsigned long ulBase,
                 unsigned long ulEndpoint,
                 unsigned long *pulFIFOAddress,
                 unsigned long *pulFIFOSize,
                 unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

pulFIFOAddress is the starting address for the FIFO.

pulFIFOSize is the size of the FIFO in bytes.

ulFlags specifies what information to retrieve from the FIFO configuration.

Description:

This function will return the starting address and size of the FIFO for a given endpoint. Endpoint zero does not have a dynamically configurable FIFO so this function should not be called for endpoint zero. The *ulFlags* parameter specifies whether the endpoint's OUT or IN FIFO should be read. If in host mode, the *ulFlags* parameter should be **USB_EP_HOST_OUT** or **USB_EP_HOST_IN**, and if in device mode the *ulFlags* parameter should be either **USB_EP_DEV_OUT** or **USB_EP_DEV_IN**.

Returns:

None.

24.3.2.19 USBFIFOConfigSet

Sets the FIFO configuration for an endpoint.

Prototype:

```
void
USBFIFOConfigSet(unsigned long ulBase,
                 unsigned long ulEndpoint,
                 unsigned long ulFIFOAddress,
                 unsigned long ulFIFOSize,
                 unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFIFOAddress is the starting address for the FIFO.

ulFIFOSize is the size of the FIFO in bytes.

ulFlags specifies what information to set in the FIFO configuration.

Description:

This function will set the starting FIFO RAM address and size of the FIFO for a given endpoint. Endpoint zero does not have a dynamically configurable FIFO so this function should not be called for endpoint zero. The *ulFIFOSize* parameter should be one of the values in the **USB_FIFO_SZ_** values. If the endpoint is going to use double buffering it should use the values with the **_DB** at the end of the value. For example, use **USB_FIFO_SZ_16_DB** to configure an endpoint to have a 16 byte double buffered FIFO. If a double buffered FIFO is used, then the actual size of the FIFO will be twice the size indicated by the *ulFIFOSize* parameter. This means that the **USB_FIFO_SZ_16_DB** value will use 32 bytes of the USB controller's FIFO memory.

The *ulFIFOAddress* value should be a multiple of 8 bytes and directly indicates the starting address in the USB controller's FIFO RAM. For example, a value of 64 indicates that the FIFO should start 64 bytes into the USB controller's FIFO memory. The *ulFlags* value specifies whether the endpoint's OUT or IN FIFO should be configured. If in host mode, use **USB_EP_HOST_OUT** or **USB_EP_HOST_IN**, and if in device mode use **USB_EP_DEV_OUT** or **USB_EP_DEV_IN**.

Returns:

None.

24.3.2.20 USBFIFOFlush

Forces a flush of an endpoint's FIFO.

Prototype:

```
void
USBFIFOFlush(unsigned long ulBase,
             unsigned long ulEndpoint,
             unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFlags specifies if the IN or OUT endpoint should be accessed.

Description:

This function will force the controller to flush out the data in the FIFO. The function can be called with either host or device controllers and requires the *ulFlags* parameter be one of **USB_EP_HOST_OUT**, **USB_EP_HOST_IN**, **USB_EP_DEV_OUT**, or **USB_EP_DEV_IN**.

Returns:

None.

24.3.2.21 USBFrameNumberGet

Get the current frame number.

Prototype:

```
unsigned long
USBFrameNumberGet(unsigned long ulBase)
```

Parameters:

ulBase specifies the USB module base address.

Description:

This function returns the last frame number received.

Returns:

The last frame number received.

24.3.2.22 USBHostAddrGet

Gets the current functional device address for an endpoint.

Prototype:

```
unsigned long
USBHostAddrGet(unsigned long ulBase,
               unsigned long ulEndpoint,
               unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFlags determines if this is an IN or an OUT endpoint.

Description:

This function returns the current functional address that an endpoint is using to communicate with a device. The *ulFlags* parameter determines if the IN or OUT endpoint's device address is returned.

Note:

This function should only be called in host mode.

Returns:

Returns the current function address being used by an endpoint.

24.3.2.23 USBHostAddrSet

Sets the functional address for the device that is connected to an endpoint in host mode.

Prototype:

```
void
USBHostAddrSet(unsigned long ulBase,
               unsigned long ulEndpoint,
```

```
                    unsigned long ulAddr,
                    unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulAddr is the functional address for the controller to use for this endpoint.

ulFlags determines if this is an IN or an OUT endpoint.

Description:

This function will set the functional address for a device that is using this endpoint for communication. This *ulAddr* parameter is the address of the target device that this endpoint will be used to communicate with. The *ulFlags* parameter indicates if the IN or OUT endpoint should be set.

Note:

This function should only be called in host mode.

Returns:

None.

24.3.2.24 USBHostEndpointConfig

Sets the base configuration for a host endpoint.

Prototype:

```
void
USBHostEndpointConfig(unsigned long ulBase,
                      unsigned long ulEndpoint,
                      unsigned long ulMaxPayload,
                      unsigned long ulNAKPollInterval,
                      unsigned long ulTargetEndpoint,
                      unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulMaxPayload is the maximum payload for this endpoint.

ulNAKPollInterval is the either the NAK timeout limit or the polling interval depending on the type of endpoint.

ulTargetEndpoint is the endpoint that the host endpoint is targeting.

ulFlags are used to configure other endpoint settings.

Description:

This function will set the basic configuration for the transmit or receive portion of an endpoint in host mode. The *ulFlags* parameter determines some of the configuration while the other parameters provide the rest. The *ulFlags* parameter determines whether this is an IN endpoint (USB_EP_HOST_IN or USB_EP_DEV_IN) or an OUT endpoint (USB_EP_HOST_OUT or USB_EP_DEV_OUT), whether this is a Full speed endpoint (USB_EP_SPEED_FULL) or a Low speed endpoint (USB_EP_SPEED_LOW).

The **USB_EP_MODE_** flags control the type of the endpoint.

- **USB_EP_MODE_CTRL** is a control endpoint.
- **USB_EP_MODE_ISOC** is an isochronous endpoint.
- **USB_EP_MODE_BULK** is a bulk endpoint.
- **USB_EP_MODE_INT** is an interrupt endpoint.

The *ulNAKPollInterval* parameter has different meanings based on the **USB_EP_MODE** value and whether or not this call is being made for endpoint zero or another endpoint. For endpoint zero or any Bulk endpoints, this value always indicates the number of frames to allow a device to NAK before considering it a timeout. If this endpoint is an isochronous or interrupt endpoint, this value is the polling interval for this endpoint.

For interrupt endpoints the polling interval is simply the number of frames between polling an interrupt endpoint. For isochronous endpoints this value represents a polling interval of 2 ^ (*ulNAKPollInterval* - 1) frames. When used as a NAK timeout, the *ulNAKPollInterval* value specifies 2 ^ (*ulNAKPollInterval* - 1) frames before issuing a time out. There are two special time out values that can be specified when setting the *ulNAKPollInterval* value. The first is **MAX_NAK_LIMIT** which is the maximum value that can be passed in this variable. The other is **DISABLE_NAK_LIMIT** which indicates that there should be no limit on the number of NAKs.

The **USB_EP_DMA_MODE_** flags enables the type of DMA used to access the endpoint's data FIFOs. The choice of the DMA mode depends on how the DMA controller is configured and how it is being used. See the "Using USB with the uDMA Controller" section for more information on DMA configuration.

When configuring the OUT portion of an endpoint, the **USB_EP_AUTO_SET** bit is specified to cause the transmission of data on the USB bus to start as soon as the number of bytes specified by *ulMaxPayload* have been written into the OUT FIFO for this endpoint.

When configuring the IN portion of an endpoint, the **USB_EP_AUTO_REQUEST** bit can be specified to trigger the request for more data once the FIFO has been drained enough to fit *ulMaxPayload* bytes. The **USB_EP_AUTO_CLEAR** bit can be used to clear the data packet ready flag automatically once the data has been read from the FIFO. If this is not used, this flag must be manually cleared via a call to USBDevEndpointStatusClear() or USBHostEndpointStatusClear().

Note:
This function should only be called in host mode.

Returns:
None.

24.3.2.25 USBHostEndpointDataAck

Acknowledge that data was read from the given endpoint's FIFO in host mode.

Prototype:
```
void
USBHostEndpointDataAck(unsigned long ulBase,
                       unsigned long ulEndpoint)
```

Parameters:
ulBase specifies the USB module base address.
ulEndpoint is the endpoint to access.

Description:
This function acknowledges that the data was read from the endpoint's FIFO. This call is used if processing is required between reading the data and acknowledging that the data has been read.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.2.26 USBHostEndpointDataToggle

Sets the value data toggle on an endpoint in host mode.

Prototype:
```
void
USBHostEndpointDataToggle(unsigned long ulBase,
                          unsigned long ulEndpoint,
                          tBoolean bDataToggle,
                          unsigned long ulFlags)
```

Parameters:
ulBase specifies the USB module base address.
ulEndpoint specifies the endpoint to reset the data toggle.
bDataToggle specifies whether to set the state to DATA0 or DATA1.
ulFlags specifies whether to set the IN or OUT endpoint.

Description:
This function is used to force the state of the data toggle in host mode. If the value passed in the *bDataToggle* parameter is **false**, then the data toggle will be set to the DATA0 state, and if it is **true** it will be set to the DATA1 state. The *ulFlags* parameter can be **USB_EP_HOST_IN** or **USB_EP_HOST_OUT** to access the desired portion of this endpoint. The *ulFlags* parameter is ignored for endpoint zero.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.2.27 USBHostEndpointStatusClear

Clears the status bits in this endpoint in host mode.

Prototype:
```
void
USBHostEndpointStatusClear(unsigned long ulBase,
                           unsigned long ulEndpoint,
                           unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFlags are the status bits that will be cleared.

Description:

This function will clear the status of any bits that are passed in the *ulFlags* parameter. The *ulFlags* parameter can take the value returned from the USBEndpointStatus() call.

Note:

This function should only be called in host mode.

Returns:

None.

24.3.2.28 USBHostHubAddrGet

Get the current device hub address for this endpoint.

Prototype:

```
unsigned long
USBHostHubAddrGet(unsigned long ulBase,
                  unsigned long ulEndpoint,
                  unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulFlags determines if this is an IN or an OUT endpoint.

Description:

This function will return the current hub address that an endpoint is using to communicate with a device. The *ulFlags* parameter determines if the device address for the IN or OUT endpoint is returned.

Note:

This function should only be called in host mode.

Returns:

This function returns the current hub address being used by an endpoint.

24.3.2.29 USBHostHubAddrSet

Set the hub address for the device that is connected to an endpoint.

Prototype:

```
void
USBHostHubAddrSet(unsigned long ulBase,
                  unsigned long ulEndpoint,
                  unsigned long ulAddr,
                  unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

ulAddr is the hub address for the device using this endpoint.

ulFlags determines if this is an IN or an OUT endpoint.

Description:

This function will set the hub address for a device that is using this endpoint for communication. The *ulFlags* parameter determines if the device address for the IN or the OUT endpoint is set by this call.

Note:

This function should only be called in host mode.

Returns:

None.

24.3.2.30 USBHostPwrDisable

Disables the external power pin.

Prototype:

```
void
USBHostPwrDisable(unsigned long ulBase)
```

Parameters:

ulBase specifies the USB module base address.

Description:

This function disables the USBEPEN signal to disable an external power supply in host mode operation.

Note:

This function should only be called in host mode.

Returns:

None.

24.3.2.31 USBHostPwrEnable

Enables the external power pin.

Prototype:

```
void
USBHostPwrEnable(unsigned long ulBase)
```

Parameters:

ulBase specifies the USB module base address.

Description:

This function enables the USBEPEN signal to enable an external power supply in host mode operation.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.2.32 USBHostPwrFaultConfig

Sets the configuration for USB power fault.

Prototype:
```
void
USBHostPwrFaultConfig(unsigned long ulBase,
                      unsigned long ulFlags)
```

Parameters:
ulBase specifies the USB module base address.
ulFlags specifies the configuration of the power fault.

Description:
This function will set the behavior of the USB controller during a power fault and the behavior of the USBPEN pin. The flags specify the power fault level sensitivity, the power fault action, and the power enable level and source. One of the following can be selected as the power fault level sensitivity:

- **USB_HOST_PWRFLT_LOW** - Power fault is indicated by the pin being driven low.
- **USB_HOST_PWRFLT_HIGH** - Power fault is indicated by the pin being driven! high.

One of the following can be selected as the power fault action:

- **USB_HOST_PWRFLT_EP_NONE** - No automatic action when power fault detected.
- **USB_HOST_PWRFLT_EP_TRI** - Automatically Tri-state the USBEPEN pin on a power fault.
- **USB_HOST_PWRFLT_EP_LOW** - Automatically drive USBEPEN pin low on a power fault.
- **USB_HOST_PWRFLT_EP_HIGH** - Automatically drive USBEPEN pin high on a power fault.

One of the following can be selected as the power enable level and source:

- **USB_HOST_PWREN_LOW** - USBEPEN is driven low when power is enabled.
- **USB_HOST_PWREN_HIGH** - USBEPEN is driven high when power is enabled.
- **USB_HOST_PWREN_VBLOW** - USBEPEN is driven high when VBUS is low.
- **USB_HOST_PWREN_VBHIGH** - USBEPEN is driven high when VBUS is high.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.2.33 USBHostPwrFaultDisable

Disables power fault detection.

Prototype:

```
void
USBHostPwrFaultDisable(unsigned long ulBase)
```

Parameters:

ulBase specifies the USB module base address.

Description:

This function disables power fault detection in the USB controller.

Note:

This function should only be called in host mode.

Returns:

None.

24.3.2.34 USBHostPwrFaultEnable

Enables power fault detection.

Prototype:

```
void
USBHostPwrFaultEnable(unsigned long ulBase)
```

Parameters:

ulBase specifies the USB module base address.

Description:

This function enables power fault detection in the USB controller. If the USBPFLT pin is not in use this function should not be used.

Note:

This function should only be called in host mode.

Returns:

None.

24.3.2.35 USBHostRequestIN

Schedules a request for an IN transaction on an endpoint in host mode.

Prototype:

```
void
USBHostRequestIN(unsigned long ulBase,
                 unsigned long ulEndpoint)
```

Parameters:

ulBase specifies the USB module base address.

ulEndpoint is the endpoint to access.

Description:
This function will schedule a request for an IN transaction. When the USB device being communicated with responds the data, the data can be retrieved by calling USBEndpointDataGet() or via a DMA transfer.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.2.36 USBHostRequestStatus

Issues a request for a status IN transaction on endpoint zero.

Prototype:
```
void
USBHostRequestStatus(unsigned long ulBase)
```

Parameters:
ulBase specifies the USB module base address.

Description:
This function is used to cause a request for an status IN transaction from a device on endpoint zero. This function can only be used with endpoint zero as that is the only control endpoint that supports this ability. This is used to complete the last phase of a control transaction to a device and an interrupt will be signaled when the status packet has been received.

Returns:
None.

24.3.2.37 USBHostReset

Handles the USB bus reset condition.

Prototype:
```
void
USBHostReset(unsigned long ulBase,
             tBoolean bStart)
```

Parameters:
ulBase specifies the USB module base address.
bStart specifies whether to start or stop signaling reset on the USB bus.

Description:
When this function is called with the *bStart* parameter set to **true**, this function will cause the start of a reset condition on the USB bus. The caller should then delay at least 20ms before calling this function again with the *bStart* parameter set to **false**.

Note:
This function should only be called in host mode.

Returns:
None.

24.3.2.38 USBHostResume

Handles the USB bus resume condition.

Prototype:
```
void
USBHostResume(unsigned long ulBase,
              tBoolean bStart)
```

Parameters:
ulBase specifies the USB module base address.
bStart specifies if the USB controller is entering or leaving the resume signaling state.

Description:
When in device mode this function will bring the USB controller out of the suspend state. This call should first be made with the *bStart* parameter set to **true** to start resume signaling. The device application should then delay at least 10ms but not more than 15ms before calling this function with the *bStart* parameter set to **false**.

When in host mode this function will signal devices to leave the suspend state. This call should first be made with the *bStart* parameter set to **true** to start resume signaling. The host application should then delay at least 20ms before calling this function with the *bStart* parameter set to **false**. This will cause the controller to complete the resume signaling on the USB bus.

Returns:
None.

24.3.2.39 USBHostSpeedGet

Returns the current speed of the USB device connected.

Prototype:
```
unsigned long
USBHostSpeedGet(unsigned long ulBase)
```

Parameters:
ulBase specifies the USB module base address.

Description:
This function will return the current speed of the USB bus.

Note:
This function should only be called in host mode.

Returns:
Returns either **USB_LOW_SPEED**, **USB_FULL_SPEED**, or **USB_UNDEF_SPEED**.

24.3.2.40 USBHostSuspend

Puts the USB bus in a suspended state.

Prototype:

```
void
USBHostSuspend(unsigned long ulBase)
```

Parameters:

ulBase specifies the USB module base address.

Description:

When used in host mode, this function will put the USB bus in the suspended state.

Note:

This function should only be called in host mode.

Returns:

None.

24.3.2.41 USBIntDisable

Disables the sources for USB interrupts.

Prototype:

```
void
USBIntDisable(unsigned long ulBase,
              unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulFlags specifies which interrupts to disable.

Description:

This function will disable the USB controller from generating the interrupts indicated by the *ulFlags* parameter. There are three groups of interrupt sources, IN Endpoints, OUT Endpoints, and general status changes, specified by **USB_INT_HOST_IN**, **USB_INT_HOST_OUT**, **USB_INT_DEV_IN**, **USB_INT_DEV_OUT**, and **USB_INT_STATUS**. If **USB_INT_ALL** is specified then all interrupts will be disabled.

Returns:

None.

24.3.2.42 USBIntEnable

Enables the sources for USB interrupts.

Prototype:

```
void
USBIntEnable(unsigned long ulBase,
             unsigned long ulFlags)
```

Parameters:

ulBase specifies the USB module base address.

ulFlags specifies which interrupts to enable.

Description:

This function will enable the USB controller's ability to generate the interrupts indicated by the *ulFlags* parameter. There are three groups of interrupt sources, IN Endpoints, OUT Endpoints, and general status changes, specified by **USB_INT_HOST_IN**, **USB_INT_HOST_OUT**, **USB_INT_DEV_IN**, **USB_INT_DEV_OUT**, and **USB_STATUS**. If **USB_INT_ALL** is specified then all interrupts will be enabled.

Note:

A call must be made to enable the interrupt in the main interrupt controller to receive interrupts. The USBIntRegister() API performs this controller level interrupt enable. However if static interrupt handlers are used then then a call to IntEnable() must be made in order to allow any USB interrupts to occur.

Returns:

None.

24.3.2.43 USBIntRegister

Registers an interrupt handler for the USB controller.

Prototype:

```
void
USBIntRegister(unsigned long ulBase,
               void (*pfnHandler)(void))
```

Parameters:

ulBase specifies the USB module base address.

pfnHandler is a pointer to the function to be called when a USB interrupt occurs.

Description:

This sets the handler to be called when a USB interrupt occurs. This will also enable the global USB interrupt in the interrupt controller. The specific desired USB interrupts must be enabled via a separate call to USBIntEnable(). It is the interrupt handler's responsibility to clear the interrupt sources via a call to USBIntStatus().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

24.3.2.44 USBIntStatus

Returns the status of the USB interrupts.

Prototype:

```
unsigned long
USBIntStatus(unsigned long ulBase)
```

Parameters:
ulBase specifies the USB module base address.

Description:
This function will read the source of the interrupt for the USB controller. There are three groups of interrupt sources, IN Endpoints, OUT Endpoints, and general status changes. This call will return the current status for all of these interrupts. The bit values returned should be compared against the **USB_HOST_IN**, **USB_HOST_OUT**, **USB_HOST_EP0**, **USB_DEV_IN**, **USB_DEV_OUT**, and **USB_DEV_EP0** values.

Note:
This call will clear the source of all of the general status interrupts.

Returns:
Returns the status of the sources for the USB controller's interrupt.

24.3.2.45 USBIntUnregister

Unregisters an interrupt handler for the USB controller.

Prototype:
```
void
USBIntUnregister(unsigned long ulBase)
```

Parameters:
ulBase specifies the USB module base address.

Description:
This function unregister the interrupt handler. This function will also disable the USB interrupt in the interrupt controller.

See also:
IntRegister() for important information about registering or unregistering interrupt handlers.

Returns:
None.

24.3.2.46 USBOTGSessionRequest

Starts or ends a session.

Prototype:
```
void
USBOTGSessionRequest(unsigned long ulBase,
                     tBoolean bStart)
```

Parameters:
ulBase specifies the USB module base address.
bStart specifies if this call starts or ends a session.

Description:

This function is used in OTG mode to start a session request or end a session. If the *bStart* parameter is set to **true**, then this function start a session and if it is **false** it will end a session.

Returns:

None.

24.4 Programming Example

This example code makes the calls necessary to configure end point 1, in device mode, as a bulk IN end point. The first call configures end point 1 to have a maximum packet size of 64 bytes and makes it a bulk IN end point. The call to USBFIFOConfig() sets the starting address to 64 bytes in and 64 bytes long. It specifies **USB_EP_DEV_IN** to indicate that this is a device mode IN endpoint. The next two calls demonstrate how to fill the data FIFO for this endpoint and then have it scheduled for transmission on the USB bus. The USBEndpointDataPut() call puts data into the FIFO but does not actually start the data transmission. The USBEndpointDataSend() call will schedule the transmission to go out the next time the host controller requests data on this endpoint.

```
//
// Configure Endpoint 1.
//
USBDevEndpointConfig(USB0_BASE, USB_EP_1, 64, DISABLE_NAK_LIMIT,
    USB_EP_MODE_BULK | USB_EP_DEV_IN);

//
// Configure FIFO as a device IN endpoint FIFO starting at address 64
// and is 64 bytes in size.
//
USBFIFOConfig(USB0_BASE, USB_EP_1, 64, USB_FIFO_SZ_64, USB_EP_DEV_IN);

...

//
// Put the data in the FIFO.
//
USBEndpointDataPut(USB0_BASE, USB_EP_1, pucData, 64);

//
// Start the transmission of data.
//
USBEndpointDataSend(USB0_BASE, USB_EP_1, USB_TRANS_IN);
```

25 Watchdog Timer

25.1 Introduction

The Watchdog Timer API provides a set of functions for using the Stellaris watchdog timer modules. Functions are provided to deal with the watchdog timer interrupts, and to handle status and configuration of the watchdog timer.

The watchdog timer module's function is to prevent system hangs. The watchdog timer module consists of a 32-bit down counter, a programmable load register, interrupt generation logic, and a locking register. Once the watchdog timer has been configured, the lock register can be written to prevent the timer configuration from being inadvertently altered.

The watchdog timer can be configured to generate an interrupt to the processor upon its first timeout, and to generate a reset signal upon its second timeout. The watchdog timer module generates the first timeout signal when the 32-bit counter reaches the zero state after being enabled; enabling the counter also enables the watchdog timer interrupt. After the first timeout event, the 32-bit counter is reloaded with the value of the watchdog timer load register, and the timer resumes counting down from that value. If the timer counts down to its zero state again before the first timeout interrupt is cleared, and the reset signal has been enabled, the watchdog timer asserts its reset signal to the system. If the interrupt is cleared before the 32-bit counter reaches its second timeout, the 32-bit counter is loaded with the value in the load register, and counting resumes from that value. If the load register is written with a new value while the watchdog timer counter is counting, then the counter is loaded with the new value and continues counting.

This driver is contained in `src/watchdog.c`, with `src/watchdog.h` containing the API definitions for use by applications.

25.2 API Functions

Functions

- void WatchdogEnable (unsigned long ulBase)
- void WatchdogIntClear (unsigned long ulBase)
- void WatchdogIntEnable (unsigned long ulBase)
- void WatchdogIntRegister (unsigned long ulBase, void (*pfnHandler)(void))
- unsigned long WatchdogIntStatus (unsigned long ulBase, tBoolean bMasked)
- void WatchdogIntUnregister (unsigned long ulBase)
- void WatchdogLock (unsigned long ulBase)
- tBoolean WatchdogLockState (unsigned long ulBase)
- unsigned long WatchdogReloadGet (unsigned long ulBase)
- void WatchdogReloadSet (unsigned long ulBase, unsigned long ulLoadVal)
- void WatchdogResetDisable (unsigned long ulBase)

- void WatchdogResetEnable (unsigned long ulBase)
- tBoolean WatchdogRunning (unsigned long ulBase)
- void WatchdogStallDisable (unsigned long ulBase)
- void WatchdogStallEnable (unsigned long ulBase)
- void WatchdogUnlock (unsigned long ulBase)
- unsigned long WatchdogValueGet (unsigned long ulBase)

25.2.1 Detailed Description

The Watchdog Timer API is broken into two groups of functions: those that deal with interrupts, and those that handle status and configuration.

The Watchdog Timer interrupts are handled by the WatchdogIntRegister(), WatchdogIntUnregister(), WatchdogIntEnable(), WatchdogIntClear(), and WatchdogIntStatus() functions.

Status and configuration functions for the Watchdog Timer module are WatchdogEnable(), WatchdogRunning(), WatchdogLock(), WatchdogUnlock(), WatchdogLockState(), WatchdogReloadSet(), WatchdogReloadGet(), WatchdogValueGet(), WatchdogResetEnable(), WatchdogResetDisable(), WatchdogStallEnable(), and WatchdogStallDisable().

25.2.2 Function Documentation

25.2.2.1 WatchdogEnable

Enables the watchdog timer.

Prototype:

```
void
WatchdogEnable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the watchdog timer module.

Description:

This will enable the watchdog timer counter and interrupt.

Note:

This function will have no effect if the watchdog timer has been locked.

See also:

WatchdogLock(), WatchdogUnlock()

Returns:

None.

25.2.2.2 WatchdogIntClear

Clears the watchdog timer interrupt.

Prototype:

```
void
WatchdogIntClear(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the watchdog timer module.

Description:

The watchdog timer interrupt source is cleared, so that it no longer asserts.

Note:

Since there is a write buffer in the Cortex-M3 processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (since NVIC still sees the interrupt source asserted).

Returns:

None.

25.2.2.3 WatchdogIntEnable

Enables the watchdog timer interrupt.

Prototype:

```
void
WatchdogIntEnable(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the watchdog timer module.

Description:

Enables the watchdog timer interrupt.

Note:

This function will have no effect if the watchdog timer has been locked.

See also:

WatchdogLock(), WatchdogUnlock(), WatchdogEnable()

Returns:

None.

25.2.2.4 WatchdogIntRegister

Registers an interrupt handler for watchdog timer interrupt.

Prototype:

```
void
WatchdogIntRegister(unsigned long ulBase,
                    void (*pfnHandler)(void))
```

Parameters:

ulBase is the base address of the watchdog timer module.

pfnHandler is a pointer to the function to be called when the watchdog timer interrupt occurs.

Description:

This function does the actual registering of the interrupt handler. This will enable the global interrupt in the interrupt controller; the watchdog timer interrupt must be enabled via WatchdogEnable(). It is the interrupt handler's responsibility to clear the interrupt source via WatchdogIntClear().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

25.2.2.5 WatchdogIntStatus

Gets the current watchdog timer interrupt status.

Prototype:

```
unsigned long
WatchdogIntStatus(unsigned long ulBase,
                  tBoolean bMasked)
```

Parameters:

ulBase is the base address of the watchdog timer module.

bMasked is **false** if the raw interrupt status is required and **true** if the masked interrupt status is required.

Description:

This returns the interrupt status for the watchdog timer module. Either the raw interrupt status or the status of interrupt that is allowed to reflect to the processor can be returned.

Returns:

Returns the current interrupt status, where a 1 indicates that the watchdog interrupt is active, and a 0 indicates that it is not active.

25.2.2.6 WatchdogIntUnregister

Unregisters an interrupt handler for the watchdog timer interrupt.

Prototype:

```
void
WatchdogIntUnregister(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the watchdog timer module.

Description:
This function does the actual unregistering of the interrupt handler. This function will clear the handler to be called when a watchdog timer interrupt occurs. This will also mask off the interrupt in the interrupt controller so that the interrupt handler no longer is called.

See also:
IntRegister() for important information about registering interrupt handlers.

Returns:
None.

25.2.2.7 WatchdogLock

Enables the watchdog timer lock mechanism.

Prototype:
```
void
WatchdogLock(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
Locks out write access to the watchdog timer configuration registers.

Returns:
None.

25.2.2.8 WatchdogLockState

Gets the state of the watchdog timer lock mechanism.

Prototype:
```
tBoolean
WatchdogLockState(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
Returns the lock state of the watchdog timer registers.

Returns:
Returns **true** if the watchdog timer registers are locked, and **false** if they are not locked.

25.2.2.9 WatchdogReloadGet

Gets the watchdog timer reload value.

Prototype:

```
unsigned long
WatchdogReloadGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the watchdog timer module.

Description:

This function gets the value that is loaded into the watchdog timer when the count reaches zero for the first time.

See also:

WatchdogReloadSet()

Returns:

None.

25.2.2.10 WatchdogReloadSet

Sets the watchdog timer reload value.

Prototype:

```
void
WatchdogReloadSet(unsigned long ulBase,
                  unsigned long ulLoadVal)
```

Parameters:

ulBase is the base address of the watchdog timer module.

ulLoadVal is the load value for the watchdog timer.

Description:

This function sets the value to load into the watchdog timer when the count reaches zero for the first time; if the watchdog timer is running when this function is called, then the value will be immediately loaded into the watchdog timer counter. If the *ulLoadVal* parameter is 0, then an interrupt is immediately generated.

Note:

This function will have no effect if the watchdog timer has been locked.

See also:

WatchdogLock(), WatchdogUnlock(), WatchdogReloadGet()

Returns:

None.

25.2.2.11 WatchdogResetDisable

Disables the watchdog timer reset.

Prototype:

```
void
WatchdogResetDisable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
Disables the capability of the watchdog timer to issue a reset to the processor upon a second timeout condition.

Note:
This function will have no effect if the watchdog timer has been locked.

See also:
WatchdogLock(), WatchdogUnlock()

Returns:
None.

25.2.2.12 WatchdogResetEnable

Enables the watchdog timer reset.

Prototype:
```
void
WatchdogResetEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
Enables the capability of the watchdog timer to issue a reset to the processor upon a second timeout condition.

Note:
This function will have no effect if the watchdog timer has been locked.

See also:
WatchdogLock(), WatchdogUnlock()

Returns:
None.

25.2.2.13 WatchdogRunning

Determines if the watchdog timer is enabled.

Prototype:
```
tBoolean
WatchdogRunning(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
This will check to see if the watchdog timer is enabled.

Returns:
Returns **true** if the watchdog timer is enabled, and **false** if it is not.

25.2.2.14 WatchdogStallDisable

Disables stalling of the watchdog timer during debug events.

Prototype:
```
void
WatchdogStallDisable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
This function disables the debug mode stall of the watchdog timer. By doing so, the watchdog timer continues to count regardless of the processor debug state.

Returns:
None.

25.2.2.15 WatchdogStallEnable

Enables stalling of the watchdog timer during debug events.

Prototype:
```
void
WatchdogStallEnable(unsigned long ulBase)
```

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
This function allows the watchdog timer to stop counting when the processor is stopped by the debugger. By doing so, the watchdog is prevented from expiring (typically almost immediately from a human time perspective) and resetting the system (if reset is enabled). The watchdog will instead expired after the appropriate number of processor cycles have been executed while debugging (or at the appropriate time after the processor has been restarted).

Returns:
None.

25.2.2.16 WatchdogUnlock

Disables the watchdog timer lock mechanism.

Prototype:

```
void
WatchdogUnlock(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the watchdog timer module.

Description:

Enables write access to the watchdog timer configuration registers.

Returns:

None.

25.2.2.17 WatchdogValueGet

Gets the current watchdog timer value.

Prototype:

```
unsigned long
WatchdogValueGet(unsigned long ulBase)
```

Parameters:

ulBase is the base address of the watchdog timer module.

Description:

This function reads the current value of the watchdog timer.

Returns:

Returns the current value of the watchdog timer.

25.3 Programming Example

The following example shows how to set up the watchdog timer API to reset the processor after two timeouts.

```
//
// Check to see if the registers are locked, and if so, unlock them.
//
if(WatchdogLockState(WATCHDOG_BASE) == true)
{
    WatchdogUnlock(WATCHDOG_BASE);
}

//
// Initialize the watchdog timer.
//
WatchdogReloadSet(WATCHDOG_BASE, 0xFEEFEE);

//
// Enable the reset.
//
WatchdogResetEnable(WATCHDOG_BASE);

//
```

```
// Enable the watchdog timer.
//
WatchdogEnable(WATCHDOG_BASE);

//
// Wait for the reset to occur.
//
while(1)
{
}
```

26 Using the ROM

26.1 Introduction

Stellaris DustDevil-class devices have portions of the peripheral driver library stored in an on-chip ROM. By utilizing the code in the on-chip ROM, more flash is available for use by the application. The boot loader is also contained within the ROM, which can be called by an application in order to start a firmware update.

26.2 Direct ROM Calls

In order to call the ROM, the following steps must be performed:

- The device on which the application will be run must be defined. This is done by defining a preprocessor symbol, which can be done either within the source code or in the project that builds the application. The later is more flexible if code is shared between projects.
- `src/rom.h` is included by the source code desiring to call the ROM.
- The ROM version of a peripheral driver library function is called. For example, if GPIODirModeSet() is to be called in the ROM, ROM_GPIODirModeSet() is used instead.

A define is used to to select the device being used since the set of functions available in the ROM must be a compile-time decision; checking at run-time does not provide any flash savings since both the ROM call and the flash version of the API would be in the application flash image.

The following define is recognized by `src/rom.h`:

`TARGET_IS_DUSTDEVIL_RA0`	The application is being built to run on a DustDevil-class device, silicon revision A0.

By using ROM_Function(), the ROM will be explicitly called. If the function in question is not available in the ROM, a compiler error will be produced.

See the *Stellaris ROM User's Guide* for details of the APIs available in the ROM.

The following is an example of calling a function in the ROM, defining the device in question using a `#define` in the source instead of in the project file:

```
#define TARGET_IS_DUSTDEVIL_RA0
#include "../src/rom.h"
#include "../src/systick.h"

int
```

```
main(void)
{
    ROM_SysTickPeriodSet(0x1000);
    ROM_SysTickEnable();

    // ...
}
```

26.3 Mapped ROM Calls

When code is intended to be shared between projects, and some of the projects run on devices with a ROM and some run on devices without a ROM, it is convenient to have the code automatically call the ROM or the flash version of the API without having `#ifdef`-s in the code. `rom_map.h` provides an automatic mapping feature for accessing the ROM. Similar to the ROM_Function() APIs provided by `rom.h`, a set of MAP_Function() APIs are provided. If the function is available in ROM, MAP_Function() will simply call ROM_Function(); otherwise it will call Function().

In order to use the mapped ROM calls, the following steps must be performed:

- Follow the above steps for including and using `src/rom.h`.
- Include `src/rom_map.h`.
- Continuing the above example, call MAP_GPIODirModeSet() in the source code.

As in the direct ROM call method, the choice of calling ROM versus the flash version is made at compile-time. The only APIs that are provided via the ROM mapping feature are ones that are available in the ROM, which is not every API available in the peripheral driver library.

The following is an example of calling a function in shared code, where the device in question is defined in the project file:

```
#include "../src/rom.h"
#include "../src/rom_map.h"
#include "../src/systick.h"

void
SetupSysTick(void)
{
    MAP_SysTickPeriodSet(0x1000);
    Map_SysTickEnable();
}
```

When built for a device that does not have a ROM, this is equivalent to:

```
#include "../src/systick.h"

void
SetupSysTick(void)
{
    SysTickPeriodSet(0x1000);
    SysTickEnable();
}
```

When built for a device that has a ROM, however, this is eqivalent to:

```
#include "../src/rom.h"
#include "../src/systick.h"

void
SetupSysTick(void)
{
    ROM_SysTickPeriodSet(0x1000);
    ROM_SysTickEnable();
}
```

26.4 Firmware Update

Functions

- void UpdateI2C (void)
- void UpdateSSI (void)
- void UpdateUART (void)

26.4.1 Detailed Description

There are a set of APIs in the ROM for restarting the boot loader in order to commence a firmware update. Multiple calls are provided since each selects a particular interface to be used for the update process, bypassing the interface selection step of the normal boot loader (including the auto-bauding in the UART interface).

See the *Stellaris ROM User's Guide* for details of the firmware update APIs in the ROM.

26.4.2 Function Documentation

26.4.2.1 UpdateI2C

Starts an update over the I2C0 interface.

Prototype:

```
void
UpdateI2C(void)
```

Description:

Calling this function commences an update of the firmware via the I2C0 interface. This function assumes that the I2C0 interface has already been configured and is currently operational. The I2C0 slave is used for data transfer, and the I2C0 master is used to monitor bus busy conditions (therefore, both must be enabled).

Returns:

Never returns.

26.4.2.2 UpdateSSI

Starts an update over the SSI0 interface.

Prototype:

```
void
UpdateSSI(void)
```

Description:

Calling this function commences an update of the firmware via the SSI0 interface. This function assumes that the SSI0 interface has already been configured and is currently oprational.

Returns:

Never returns.

26.4.2.3 UpdateUART

Starts an update over the UART0 interface.

Prototype:

```
void
UpdateUART(void)
```

Description:

Calling this function commences an update of the firmware via the UART0 interface. This function assumes that the UART0 interface has already been configured and is currently operational.

Returns:

Never returns.

27 Utility Functions

27.1 Introduction

The utility functions are a loose collection of functions that are not specific to any Stellaris peripheral or board. These functions provide mechanisms for communicating with the debugger and for providing a simple serial terminal on one of the UARTs. There are also lightweight implementations of functions for printf style formatted output.

27.2 API Functions

Functions

- int CmdLineProcess (char *pcCmdLine)
- int DiagClose (int iHandle)
- char * DiagCommandString (char *pcBuf, unsigned long ulLen)
- void DiagExit (int iRet)
- long DiagFlen (int iHandle)
- int DiagOpen (const char *pcName, int iMode)
- int DiagOpenStdio (void)
- void DiagPrintf (int iHandle, const char *pcString,...)
- int DiagRead (int iHandle, char *pcBuf, unsigned long ulLen, int iMode)
- int DiagWrite (int iHandle, const char *pcBuf, unsigned long ulLen, int iMode)
- unsigned char * FlashPBGet (void)
- void FlashPBInit (unsigned long ulStart, unsigned long ulEnd, unsigned long ulSize)
- void FlashPBSave (unsigned char *pucBuffer)
- void lwIPEthernetIntHandler (void)
- void lwIPInit (const unsigned char *pucMAC, unsigned long ulIPAddr, unsigned long ulNetMask, unsigned long ulGWAddr, unsigned long ulIPMode)
- unsigned long lwIPLocalGWAddrGet (void)
- unsigned long lwIPLocalIPAddrGet (void)
- void lwIPLocalMACGet (unsigned char *pucMAC)
- unsigned long lwIPLocalNetMaskGet (void)
- void lwIPNetworkConfigChange (unsigned long ulIPAddr, unsigned long ulNetMask, unsigned long ulGWAddr, unsigned long ulIPMode)
- void lwIPTimer (unsigned long ulTimeMS)
- void RingBufAdvanceRead (tRingBufObject *ptRingBuf, unsigned long ulNumBytes)
- void RingBufAdvanceWrite (tRingBufObject *ptRingBuf, unsigned long ulNumBytes)
- unsigned long RingBufContigFree (tRingBufObject *ptRingBuf)
- unsigned long RingBufContigUsed (tRingBufObject *ptRingBuf)

- tBoolean RingBufEmpty (tRingBufObject *ptRingBuf)
- void RingBufFlush (tRingBufObject *ptRingBuf)
- unsigned long RingBufFree (tRingBufObject *ptRingBuf)
- tBoolean RingBufFull (tRingBufObject *ptRingBuf)
- void RingBufInit (tRingBufObject *ptRingBuf, unsigned char *pucBuf, unsigned long ulSize)
- void RingBufRead (tRingBufObject *ptRingBuf, unsigned char *pucData, unsigned long ulLength)
- unsigned char RingBufReadOne (tRingBufObject *ptRingBuf)
- unsigned long RingBufSize (tRingBufObject *ptRingBuf)
- unsigned long RingBufUsed (tRingBufObject *ptRingBuf)
- void RingBufWrite (tRingBufObject *ptRingBuf, unsigned char *pucData, unsigned long ulLength)
- void RingBufWriteOne (tRingBufObject *ptRingBuf, unsigned char ucData)
- void SoftwareUpdateBegin (void)
- void SoftwareUpdateInit (tSoftwareUpdateRequested pfnCallback)
- void UARTFlushRx (void)
- void UARTFlushTx (tBoolean bDiscard)
- int UARTgets (char *pcBuf, unsigned long ulLen)
- int UARTPeek (unsigned char ucChar)
- void UARTprintf (const char *pcString,...)
- void UARTStdioInit (unsigned long ulPortNum)
- void ulocaltime (unsigned long ulTime, tTime *psTime)
- int usnprintf (char *pcBuf, unsigned long ulSize, const char *pcString,...)
- int usprintf (char *pcBuf, const char *pcString,...)
- char * ustrstr (const char *pcHaystack, const char *pcNeedle)
- unsigned long ustrtoul (const char *pcStr, const char **ppcStrRet, int iBase)
- int uvsnprintf (char *pcBuf, unsigned long ulSize, const char *pcString, va_list vaArgP)

27.2.1 Detailed Description

The first group of utility functions are the diagnostic functions (“Diag”) used for interacting with the debugger (if the debugger supports that feature). The Diag functions allow the program to open a handle on the file system of the host system, allowing reading or writing of files, or communication with a console. This feature is sometimes referred to as “semihosting”. Not all debuggers support all of these features. The debugger specific support is provided in a source file that is linked into the application. The source files for debugger support can be found in the “utils” directory.

The Diag functions are used as follows: DiagOpen() is used to open a file on the host system. DiagOpenStdio() is used to open a handle to the console, typically for displaying messages to the user. Once a handle is opened, it can be closed with DiagClose(). The functions DiagRead() and DiagWrite() are used for reading and writing to the host, while DiagPrintf() provides printf style formatting of output. DiagFlen() is used to find the size of a file, DiagCommandString() used to get the command line arguments from the debugger, and DiagExit() to exit the program and return control to the debugger.

The second group of utility functions is used to provide a simple UART based console. UARTStdioInit() is used to initialize a specific UART to be used as the console serial port. Then the function UARTprintf() can be used to send formatted output to the serial port, which UARTgets() can be

used to get a line of input from the serial console. By default, the uartstdio module operates in a blocking mode. To allow non-blocking operation in which transmit and receive buffers are used and data transfer is managed under interrupt control, a build-time switch (**UART_BUFFERED**) may be defined.

The above functions, along with CmdLineProcess() can be used to implement a simple command line processor. The function CmdLineProcess() will break up any command line in a buffer into command line arguments in "argc, argv" form, match the first argument to a command name in a command table, and then call the function that implements the command.

The third group of functions are used for providing simple forms of some standard library string formatting functions. If the simplified functions meet the needs of the application for formatted output, then they may be used in place of the equivalent standard library functions, which may save on overall code size.

The library formatting functions are usprintf() which is a simplified replacement for sprintf(). Likewise usnprintf() and uvsnprintf() are simplified replacements for snprintf() and vsnprintf() from the standard library. These functions are lighter weight than the equivalent library function because they offer simpler and reduced conversion options, and elimination of floating point support. If these features are needed then the standard library can still be used.

The fourth group of functions are used for providing a simple, fault-tolerant, persistent storage mechanism for storing parameter information for an application.

The FlashPBInit() function is used to initialize a parameter block. The primary conditions for the parameter block are that flash region used to store the parameter blocks must contain at least two (2) erase blocks of flash to ensure fault tolerance, and the size of the parameter block must be an integral divisor of the the size of an erase block. FlashPBGet() and FlashPBSave() are used to read and write parameter block data into the parameter region. The only constaints on the content of the parameter block are that the first two bytes of the block are reserved for use by the read/write functions as a sequence number and checksum, respectively.

The fifth group of functions are used to provide a simple abstraction layer for the lwIP version 1.3.0 TCP/IP stack.

The lwIPInit() function is used to initialize the lwIP TCP/IP stack, based on the options defined in lwipopts.h. The lwIPEthernetIntHandler() is the interrupt handler function for use with the lwIP TCP/IP stack. This handler will process transmit and receive packets. If no RTOS is being used, the interrupt handler will also service the lwIP timers. The lwIPTimer() function is to be called periodically to support the TCP, ARP, DHCP and other timers used by the lwIP TCP/IP stack. If no RTOS is being used, this timer function will simply trigger an Ethernet interrupt to allow the interrupt handler to service the timers.

Refer to the individual function documentation for the name of it's source and header files.

27.2.2 Function Documentation

27.2.2.1 CmdLineProcess

Process a command line string into arguments and execute the command.

Prototype:

```
int
CmdLineProcess(char *pcCmdLine)
```

Parameters:

pcCmdLine points to a string that contains a command line that was obtained by an application by some means.

Description:

This function will take the supplied command line string and break it up into individual arguments. The first argument is treated as a command and is searched for in the command table. If the command is found, then the command function is called and all of the command line arguments are passed in the normal argc, argv form.

The command table is contained in an array named g_sCmdTable which must be implemented in the application.

This function is contained in `utils/cmdline.c`, with `utils/cmdline.h` containing the API definition for use by applications.

Returns:

Returns **CMDLINE_BAD_CMD** if the command is not found, **CMDLINE_TOO_MANY_ARGS** if there are more arguments than can be parsed. Otherwise it returns the code that was returned by the command function.

27.2.2.2 DiagClose

Closes a host file system file.

Prototype:

```
int
DiagClose(int iHandle)
```

Parameters:

iHandle is the handle of the file to close.

Description:

This function closes a file previously opened with DiagOpen(); this is similar to the `fclose()` function in the C library.

This function is contained in a debugger-specific `utils/<debugger>.?`, with `utils/diag.h` containing the API definition for use by applications.

Returns:

Returns zero on success or non-zero on failure.

27.2.2.3 DiagCommandString

Gets the command line arguments from the debugger.

Prototype:

```
char *
DiagCommandString(char *pcBuf,
                  unsigned long ulLen)
```

Parameters:

pcBuf is a pointer to the buffer to be filled with the command line arguments.

ulLen is the length of the buffer.

Description:

This function retrieves the command line arguments from the debugger, if it is able to provide them. The raw command line string is returned; it is the responsibility of the application to parse it into an `argc`/`argv` pair if desired.

This function is contained in a debugger-specific `utils/<debugger>.?`, with `utils/diag.h` containing the API definition for use by applications.

Returns:

Returns a pointer to the returned command line (typically the same as the supplied buffer) on success and NULL if the command line is not available.

27.2.2.4 DiagExit

Terminates the application.

Prototype:

```
void
DiagExit(int iRet)
```

Parameters:

iRet is the return value from the application.

Description:

This function terminates the application; this is similar to the `exit()` function in the C library.

This function is contained in a debugger-specific `utils/<debugger>.?`, with `utils/diag.h` containing the API definition for use by applications.

Returns:

Does not return.

27.2.2.5 DiagFlen

Gets the length of a host file system file.

Prototype:

```
long
DiagFlen(int iHandle)
```

Parameters:

iHandle is the handle of the file to query.

Description:

This function determines the length of a file previously opened with DiagOpen(); this is similar to seeking to the end of the file with the `fseek()` function and then doing an `ftell()`, except that the file pointer is not moved.

This function is contained in a debugger-specific `utils/<debugger>.?`, with `utils/diag.h` containing the API definition for use by applications.

Returns:
Returns the number of bytes in the file.

27.2.2.6 DiagOpen

Opens a host file system file.

Prototype:
```
int
DiagOpen(const char *pcName,
         int iMode)
```

Parameters:
pcName is the name of the file to be opened.
iMode is the mode used to open the file.

Description:
This function opens a file on the host file system; this is similar to the `fopen()` function in the C library.

The *iMode* parameter must be the logical OR of at least one of the following values (which are analogous to the mode parameter of the C library `fopen()` function):

- **OPEN_R** to open the file for reading.
- **OPEN_W** to open the file for writing.
- **OPEN_A** to append to the end of the file.
- **OPEN_B** to access the file in binary mode, which means that no end of line translations are made.
- **OPEN_PLUS** to open the file for reading and writing.

This function is contained in a debugger-specific `utils/<debugger>.?`, with `utils/diag.h` containing the API definition for use by applications.

Returns:
Returns a positive number on success or -1 on failure.

27.2.2.7 DiagOpenStdio

Opens a handle for stdio functions (both stdin and stdout).

Prototype:
```
int
DiagOpenStdio(void)
```

Description:
This function opens a handle for interacting with the user via the debugger (similar to stdin and stdout). This handle should be passed to DiagRead() to get input from the user and to DiagWrite() to display messages to the user (such as via DiagPrintf()).

This function is contained in a debugger-specific `utils/<debugger>.?`, with `utils/diag.h` containing the API definition for use by applications.

Returns:
Returns a positive number on success or -1 on failure.

27.2.2.8 DiagPrintf

A simple diagnostic printf function supporting %c, %d, %p, %s, %u, %x, and %X.

Prototype:
```
void
DiagPrintf(int iHandle,
           const char *pcString,
           ...)
```

Parameters:
iHandle is the handle of the stream to which the string is written.
pcString is the format string.
... are the optional arguments, which depend on the contents of the format string.

Description:
This function is very similar to the C library `fprintf()` function. All of its output will be sent to DiagWrite() using the supplied handle. Only the following formatting characters are supported:

- %c to print a character
- %d to print a decimal value
- %s to print a string
- %u to print an unsigned decimal value
- %x to print a hexadecimal value using lower case letters
- %X to print a hexadecimal value using lower case letters (not upper case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %s, %d, %u„ %p, %x, and %X, an optional number may reside between the % and the format character, which specifies the minimum number of characters to use for that value; if preceeded by a 0 then the extra characters will be filled with zeros instead of spaces. For example, "%8d" will use eight characters to print the decimal value with spaces added to reach eight; "%08d" will use eight characters as well but will add zeroes instead of spaces.

The type of the arguments after *pcString* must match the requirements of the format string. For example, if an integer was passed where a string was expected, an error of some kind will most likely occur.

This function is contained in `utils/diagprintf.c`, with `utils/diagprintf.h` containing the API definition for use by applications.

Returns:
None.

27.2.2.9 DiagRead

Reads data from a host file system file.

Prototype:

```
int
DiagRead(int iHandle,
         char *pcBuf,
         unsigned long ulLen,
         int iMode)
```

Parameters:

iHandle is the handle of the file to read.

pcBuf is a pointer to the buffer to contain the data read.

ulLen is the number of bytes to read from the file.

iMode is the mode used to open the file.

Description:

This function reads data from a file previously opened with DiagOpen(); this is similar to the `fread()` function in the C library.

The *iMode* parameter might be used in some debugger interfaces to adjust how the data is read from the file. Unexpected results may occur if the same value passed to DiagOpen() is not passed to DiagRead().

This function is contained in a debugger-specific `utils/<debugger>.?`, with `utils/diag.h` containing the API definition for use by applications.

Returns:

Returns zero on success, a positive number to indicate the number of bytes not read, a number with the MSB set to indicate the number of bytes not read and that EOF was encountered, or -1 to indicate an error.

27.2.2.10 DiagWrite

Writes data to a host file system file.

Prototype:

```
int
DiagWrite(int iHandle,
          const char *pcBuf,
          unsigned long ulLen,
          int iMode)
```

Parameters:

iHandle is the handle of the file to which to write.

pcBuf is a pointer to the data to be written.

ulLen is the number of bytes to write to the file.

iMode is the mode used to open the file.

Description:

This function writes data to a file previously opened with DiagOpen(); this is similar to the `fwrite()` function in the C library.

The *iMode* parameter might be used in some debugger interfaces to adjust how the data is written to the file. Unexpected results may occur if the same value passed to DiagOpen() is not passed to DiagWrite().

This function is contained in a debugger-specific `utils/<debugger>.?`, with `utils/diag.h` containing the API definition for use by applications.

Returns:

Returns zero on success, a positive number to indicate the number of bytes not written (which is an error of sorts), or a negative number to indicate an error.

27.2.2.11 FlashPBGet

Gets the address of the most recent parameter block.

Prototype:

```
unsigned char *
FlashPBGet(void)
```

Description:

This function returns the address of the most recent parameter block that is stored in flash.

This function is contained in `utils/flash_pb.c`, with `utils/flash_pb.h` containing the API definition for use by applications.

Returns:

Returns the address of the most recent parameter block, or NULL if there are no valid parameter blocks in flash.

27.2.2.12 FlashPBInit

Initializes the flash parameter block.

Prototype:

```
void
FlashPBInit(unsigned long ulStart,
            unsigned long ulEnd,
            unsigned long ulSize)
```

Parameters:

ulStart is the address of the flash memory to be used for storing flash parameter blocks; this must be the start of an erase block in the flash.

ulEnd is the address of the end of flash memory to be used for storing flash parameter blocks; this must be the start of an erase block in the flash (the first block that is NOT part of the flash memory to be used), or the address of the first word after the flash array if the last block of flash is to be used.

ulSize is the size of the parameter block when stored in flash; this must be a power of two less than or equal to the flash erase block size (typically 1024).

Description:

This function initializes a fault-tolerant, persistent storage mechanism for a parameter block for an application. The last several erase blocks of flash (as specified by *ulStart* and *ulEnd* are used for the storage; more than one erase block is required in order to be fault-tolerant.

A parameter block is an array of bytes that contain the persistent parameters for the application. The only special requirement for the parameter block is that the first byte is a sequence

number (explained in FlashPBSave()) and the second byte is a checksum used to validate the correctness of the data (the checksum byte is the byte such that the sum of all bytes in the parameter block is zero).

The portion of flash for parameter block storage is split into N equal-sized regions, where each region is the size of a parameter block (*ulSize*). Each region is scanned to find the most recent valid parameter block. The region that has a valid checksum and has the highest sequence number (with special consideration given to wrapping back to zero) is considered to be the current parameter block.

In order to make this efficient and effective, two conditions must be met. The first is *ulStart* and *ulEnd* must be specified such that at least two erase blocks of flash are dedicated to parameter block storage. If not, fault tolerance can not be guaranteed since an erase of a single block will leave a window where there are no valid parameter blocks in flash. The second condition is that the size (*ulSize*) of the parameter block must be an integral divisor of the size of an erase block of flash. If not, a parameter block will end up spanning between two erase blocks of flash, making it more difficult to manage.

When the microcontroller is initially programmed, the flash blocks used for parameter block storage are left in an erased state.

This function must be called before any other flash parameter block functions are called.

This function is contained in `utils/flash_pb.c`, with `utils/flash_pb.h` containing the API definition for use by applications.

Returns:
None.

27.2.2.13 FlashPBSave

Writes a new parameter block to flash.

Prototype:
```
void
FlashPBSave(unsigned char *pucBuffer)
```

Parameters:
pucBuffer is the address of the parameter block to be written to flash.

Description:
This function will write a parameter block to flash. Saving the new parameter blocks involves three steps:

- Setting the sequence number such that it is one greater than the sequence number of the latest parameter block in flash.
- Computing the checksum of the parameter block.
- Writing the parameter block into the storage immediately following the latest parameter block in flash; if that storage is at the start of an erase block, that block is erased first.

By this process, there is always a valid parameter block in flash. If power is lost while writing a new parameter block, the checksum will not match and the partially written parameter block will be ignored. This is what makes this fault-tolerant.

Another benefit of this scheme is that it provides wear leveling on the flash. Since multiple parameter blocks fit into each erase block of flash, and multiple erase blocks are used for parameter block storage, it takes quite a few parameter block saves before flash is re-written.

This function is contained in `utils/flash_pb.c`, with `utils/flash_pb.h` containing the API definition for use by applications.

Returns:
None.

27.2.2.14 lwIPEthernetIntHandler

Handles Ethernet interrupts for the lwIP TCP/IP stack.

Prototype:
```
void
lwIPEthernetIntHandler(void)
```

Description:
This function handles Ethernet interrupts for the lwIP TCP/IP stack. At the lowest level, all receive packets are placed into a packet queue for processing at a higher level. Also, the transmit packet queue is checked and packets are drained and transmitted through the Ethernet MAC as needed. If the system is configured without an RTOS, additional processing is performed at the interrupt level. The packet queues are processed by the lwIP TCP/IP code, and lwIP periodic timers are serviced (as needed).

Returns:
None.

27.2.2.15 lwIPInit

Initializes the lwIP TCP/IP stack.

Prototype:
```
void
lwIPInit(const unsigned char *pucMAC,
         unsigned long ulIPAddr,
         unsigned long ulNetMask,
         unsigned long ulGWAddr,
         unsigned long ulIPMode)
```

Parameters:
pucMAC is a pointer to a six byte array containing the MAC address to be used for the interface.

ulIPAddr is the IP address to be used (static).

ulNetMask is the network mask to be used (static).

ulGWAddr is the Gateway address to be used (static).

ulIPMode is the IP Address Mode. 0 will force static IP addressing to be used, 1 will force DHCP with fallback to Link Local (Auto IP), while 2 will force Link Local only.

Description:
This function performs initialization of the lwIP TCP/IP stack for the Stellaris Ethernet MAC, including DHCP and/or AutoIP, as configured.

Returns:
None.

27.2.2.16 lwIPLocalGWAddrGet

Returns the gateway address for this interface.

Prototype:
```
unsigned long
lwIPLocalGWAddrGet(void)
```

Description:
This function will read and return the currently assigned gateway address for the Stellaris Ethernet interface.

Returns:
the assigned gateway address for this interface.

27.2.2.17 lwIPLocalIPAddrGet

Returns the IP address for this interface.

Prototype:
```
unsigned long
lwIPLocalIPAddrGet(void)
```

Description:
This function will read and return the currently assigned IP address for the Stellaris Ethernet interface.

Returns:
Returns the assigned IP address for this interface.

27.2.2.18 lwIPLocalMACGet

Returns the local MAC/HW address for this interface.

Prototype:
```
void
lwIPLocalMACGet(unsigned char *pucMAC)
```

Parameters:
pucMAC is a pointer to an array of bytes used to store the MAC address.

Description:
This function will read the currently assigned MAC address into the array pased in *pucMAC*.

Returns:
None.

27.2.2.19 lwIPLocalNetMaskGet

Returns the network mask for this interface.

Prototype:
```
unsigned long
lwIPLocalNetMaskGet(void)
```

Description:
This function will read and return the currently assigned network mask for the Stellaris Ethernet interface.

Returns:
the assigned network mask for this interface.

27.2.2.20 lwIPNetworkConfigChange

Change the configuration of the lwIP network interface.

Prototype:
```
void
lwIPNetworkConfigChange(unsigned long ulIPAddr,
                        unsigned long ulNetMask,
                        unsigned long ulGWAddr,
                        unsigned long ulIPMode)
```

Parameters:
ulIPAddr is the new IP address to be used (static).
ulNetMask is the new network mask to be used (static).
ulGWAddr is the new Gateway address to be used (static).
ulIPMode is the IP Address Mode. 0 will force static IP addressing to be used, 1 will force DHCP with fallback to Link Local (Auto IP), while 2 will force Link Local only.

Description:
This function will evaluate the new configuration data. If necessary, the interface will be brought down, reconfigured, and then brought back up with the new configuration.

Returns:
None.

27.2.2.21 lwIPTimer

Handles periodic timer events for the lwIP TCP/IP stack.

Prototype:
```
void
lwIPTimer(unsigned long ulTimeMS)
```

Parameters:

ulTimeMS is the incremental time for this periodic interrupt.

Description:

This function will update the local timer by the value in *ulTimeMS*. If the system is configured for use without an RTOS, an Ethernet interrupt will be triggered to allow the lwIP periodic timers to be serviced in the Ethernet interrupt.

Returns:

None.

27.2.2.22 RingBufAdvanceRead

Remove bytes from the ring buffer by advancing the read index.

Prototype:

```
void
RingBufAdvanceRead(tRingBufObject *ptRingBuf,
                   unsigned long ulNumBytes)
```

Parameters:

ptRingBuf points to the ring buffer from which bytes are to be removed.

ulNumBytes is the number of bytes to be removed from the buffer.

Description:

This function advances the ring buffer read index by a given number of bytes, removing that number of bytes of data from the buffer. If *ulNumBytes* is larger than the number of bytes currently in the buffer, the buffer is emptied.

Returns:

None.

27.2.2.23 RingBufAdvanceWrite

Add bytes to the ring buffer by advancing the write index.

Prototype:

```
void
RingBufAdvanceWrite(tRingBufObject *ptRingBuf,
                    unsigned long ulNumBytes)
```

Parameters:

ptRingBuf points to the ring buffer to which bytes have been added.

ulNumBytes is the number of bytes added to the buffer.

Description:

This function should be used by clients who wish to add data to the buffer directly rather than via calls to RingBufWrite() or RingBufWriteOne(). It advances the write index by a given number of bytes. If the *ulNumBytes* parameter is larger than the amount of free space in the buffer, the read pointer will be advanced to cater for the addition. Note that this will result in some of the oldest data in the buffer being discarded.

Returns:
None.

27.2.2.24 RingBufContigFree

Returns number of contiguous free bytes available in a ring buffer.

Prototype:
```
unsigned long
RingBufContigFree(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to check.

Description:
This function returns the number of contiguous free bytes ahead of the current write pointer in the ring buffer.

Returns:
Returns the number of contiguous bytes available in the ring buffer.

27.2.2.25 RingBufContigUsed

Returns number of contiguous bytes of data stored in ring buffer ahead of the current read pointer.

Prototype:
```
unsigned long
RingBufContigUsed(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to check.

Description:
This function returns the number of contiguous bytes of data available in the ring buffer ahead of the current read pointer. This represents the largest block of data which does not straddle the buffer wrap.

Returns:
Returns the number of contiguous bytes available.

27.2.2.26 RingBufEmpty

Determines whether the ring buffer whose pointers and size are provided is empty or not.

Prototype:
```
tBoolean
RingBufEmpty(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to empty.

Description:
This function is used to determine whether or not a given ring buffer is empty. The structure is specifically to ensure that we do not see warnings from the compiler related to the order of volatile accesses being undefined.

Returns:
Returns **true** if the buffer is empty or **false** otherwise.

27.2.2.27 RingBufFlush

Empties the ring buffer.

Prototype:
```
void
RingBufFlush(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to empty.

Description:
Discards all data from the ring buffer.

Returns:
None.

27.2.2.28 RingBufFree

Returns number of bytes available in a ring buffer.

Prototype:
```
unsigned long
RingBufFree(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to check.

Description:
This function returns the number of bytes available in the ring buffer.

Returns:
Returns the number of bytes available in the ring buffer.

27.2.2.29 RingBufFull

Determines whether the ring buffer whose pointers and size are provided is full or not.

Prototype:
```
tBoolean
RingBufFull(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to empty.

Description:
This function is used to determine whether or not a given ring buffer is full. The structure is specifically to ensure that we do not see warnings from the compiler related to the order of volatile accesses being undefined.

Returns:
Returns **true** if the buffer is full or **false** otherwise.

27.2.2.30 RingBufInit

Initialize a ring buffer object.

Prototype:
```
void
RingBufInit(tRingBufObject *ptRingBuf,
            unsigned char *pucBuf,
            unsigned long ulSize)
```

Parameters:
ptRingBuf points to the ring buffer to be initialized.
pucBuf points to the data buffer to be used for the ring buffer.
ulSize is the size of the buffer in bytes.

Description:
This function initializes a ring buffer object, preparing it to store data.

Returns:
None.

27.2.2.31 RingBufRead

Reads data from a ring buffer.

Prototype:
```
void
RingBufRead(tRingBufObject *ptRingBuf,
            unsigned char *pucData,
            unsigned long ulLength)
```

Parameters:
ptRingBuf points to the ring buffer to be read from.
pucData points to where the data should be stored.
ulLength is the number of bytes to be read.

Description:
This function reads a sequence of bytes from a ring buffer.

Returns:
None.

27.2.2.32 RingBufReadOne

Reads a single byte of data from a ring buffer.

Prototype:

```
unsigned char
RingBufReadOne(tRingBufObject *ptRingBuf)
```

Parameters:

ptRingBuf points to the ring buffer to be written to.

Description:

This function reads a single byte of data from a ring buffer.

Returns:

The byte read from the ring buffer.

27.2.2.33 RingBufSize

Return size in bytes of a ring buffer.

Prototype:

```
unsigned long
RingBufSize(tRingBufObject *ptRingBuf)
```

Parameters:

ptRingBuf is the ring buffer object to check.

Description:

This function returns the size of the ring buffer.

Returns:

Returns the size in bytes of the ring buffer.

27.2.2.34 RingBufUsed

Returns number of bytes stored in ring buffer.

Prototype:

```
unsigned long
RingBufUsed(tRingBufObject *ptRingBuf)
```

Parameters:

ptRingBuf is the ring buffer object to check.

Description:

This function returns the number of bytes stored in the ring buffer.

Returns:

Returns the number of bytes stored in the ring buffer.

27.2.2.35 RingBufWrite

Writes data to a ring buffer.

Prototype:

```
void
RingBufWrite(tRingBufObject *ptRingBuf,
             unsigned char *pucData,
             unsigned long ulLength)
```

Parameters:

ptRingBuf points to the ring buffer to be written to.

pucData points to the data to be written.

ulLength is the number of bytes to be written.

Description:

This function write a sequence of bytes into a ring buffer.

Returns:

None.

27.2.2.36 RingBufWriteOne

Writes a single byte of data to a ring buffer.

Prototype:

```
void
RingBufWriteOne(tRingBufObject *ptRingBuf,
                unsigned char ucData)
```

Parameters:

ptRingBuf points to the ring buffer to be written to.

ucData is the byte to be written.

Description:

This function writes a single byte of data into a ring buffer.

Returns:

None.

27.2.2.37 SoftwareUpdateBegin

Passes control to the bootloader and initiates a remote software update over Ethernet.

Prototype:

```
void
SoftwareUpdateBegin(void)
```

Description:

This function passes control to the bootloader and initiates an update of the main application firmware image via BOOTP across Ethernet. This function may only be used on parts supporting Ethernet and in cases where the Ethernet boot loader is in use alongside the main application image. It must not be called in interrupt context.

Applications wishing to make use of this function must be built to operate with the bootloader. If this function is called on a system which does not include the bootloader, the results are unpredictable.

Note:

It is not safe to call this function from within the callback provided on the initial call to SoftwareUpdateInit(). The application must use the callback to signal a pending update (assuming the update is to be permitted) to some other code running in a non-interrupt context.

Returns:

Never returns.

27.2.2.38 SoftwareUpdateInit

Initializes the remote Ethernet software update notification feature.

Prototype:

```
void
SoftwareUpdateInit(tSoftwareUpdateRequested pfnCallback)
```

Parameters:

pfnCallback is a pointer to a function which will be called whenever a remote firmware update request is received. If the application wishes to allow the update to go ahead, it must call SoftwareUpdateBegin() from non-interrupt context after the callback is received. Note that the callback will most likely be made in interrupt context so it is not safe to call SoftwareUpdateBegin() from within the callback itself.

Description:

This function may be used on Ethernet-enabled parts to support remotely-signalled firmware updates over Ethernet. The LM Flash Programmer (LMFlash.exe) application sends a magic packet to UDP port 9 whenever the user requests an Ethernet-based firmware update. This packet consists of 6 bytes of 0xAA followed by the target MAC address repeated 4 times. This function starts listening on UDP port 9 and, if a magic packet matching the MAC address of this board is received, makes a call to the provided callback function to indicate that an update has been requested.

The callback function provided here will be typically be called in the context of the lwIP Ethernet interrupt handler. It is not safe to call SoftwareUpdateBegin() in this context so the application should use the callback to signal code running in a non-interrupt context to perform the update if it is to be allowed.

UDP port 9 is chosen for this function since this is the well-known port associated with "discard" operation. In other words, any other system receiving the magic packet will simply ignore it. The actual magic packet used is modeled on Wake-On-LAN which uses a similar structure (6 bytes of 0xFF followed by 16 repetitions of the target MAC address). Some Wake-On-LAN implementations also use UDP port 9 for their signalling.

Note:
Applications using this function must initialize the lwIP stack prior to making this call and must ensure that the lwIPTimer() function is called periodically. lwIP UDP must be enabled in lwipopts.h to ensure that the magic packets can be received.

Returns:
None.

27.2.2.39 UARTFlushRx

Flushes the receive buffer.

Prototype:
```
void
UARTFlushRx(void)
```

Description:
This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to discard any data received from the UART but not yet read using UARTgets().

This function is contained in `utils/uartstdio.c`, with `utils/uartstdio.h` containing the API definition for use by applications.

Returns:
None.

27.2.2.40 UARTFlushTx

Flushes the transmit buffer.

Prototype:
```
void
UARTFlushTx(tBoolean bDiscard)
```

Parameters:
bDiscard indicates whether any remaining data in the buffer should be discarded (**true**) or transmitted (**false**).

Description:
This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to flush the transmit buffer, either discarding or transmitting any data received via calls to UARTprintf() that remains untransmitted. On return, the transmit buffer will be empty.

This function is contained in `utils/uartstdio.c`, with `utils/uartstdio.h` containing the API definition for use by applications.

Returns:
None.

27.2.2.41 UARTgets

A simple UART based get string function, with some line processing.

Prototype:

```
int
UARTgets(char *pcBuf,
         unsigned long ulLen)
```

Parameters:

pcBuf points to a buffer for the incoming string from the UART.

ulLen is the length of the buffer for storage of the string, including the trailing 0.

Description:

This function will receive a string from the UART input and store the characters in the buffer pointed to by *pcBuf*. The characters will continue to be stored until a termination character is received. The termination characters are CR, LF, or ESC. A CRLF pair is treated as a single termination character. The termination characters are not stored in the string. The string will be terminated with a 0 and the function will return.

In both buffered and unbuffered modes, this function will block until a termination character is received. If non-blocking operation is required in buffered mode, a call to UARTPeek() may be made to determine whether a termination character already exists in the receive buffer prior to calling UARTgets().

Since the string will be null terminated, the user must ensure that the buffer is sized to allow for the additional null character.

This function is contained in `utils/uartstdio.c`, with `utils/uartstdio.h` containing the API definition for use by applications.

Returns:

Returns the count of characters that were stored, not including the trailing 0.

27.2.2.42 UARTPeek

Looks ahead in the receive buffer for a particular character.

Prototype:

```
int
UARTPeek(unsigned char ucChar)
```

Parameters:

ucChar is the character that is to be searched for.

Description:

This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to look ahead in the receive buffer for a particular character and report its position if found. It is typically used to determine whether a complete line of user input is available, in which case ucChar should be set to CR ('\r') which is used as the line end marker in the receive buffer.

This function is contained in `utils/uartstdio.c`, with `utils/uartstdio.h` containing the API definition for use by applications.

Returns:

Returns -1 to indicate that the requested character does not exist in the receive buffer. Returns a non-negative number if the character was found in which case the value represents the position of the first instance of *ucChar* relative to the receive buffer read pointer.

27.2.2.43 UARTprintf

A simple UART based printf function supporting %c, %d, %p, %s, %u, %x, and %X.

Prototype:

```
void
UARTprintf(const char *pcString,
           ...)
```

Parameters:

pcString is the format string.

... are the optional arguments, which depend on the contents of the format string.

Description:

This function is very similar to the C library `fprintf()` function. All of its output will be sent to the UART. Only the following formatting characters are supported:

- %c to print a character
- %d to print a decimal value
- %s to print a string
- %u to print an unsigned decimal value
- %x to print a hexadecimal value using lower case letters
- %X to print a hexadecimal value using lower case letters (not upper case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %s, %d, %u, %p, %x, and %X, an optional number may reside between the % and the format character, which specifies the minimum number of characters to use for that value; if preceeded by a 0 then the extra characters will be filled with zeros instead of spaces. For example, "%8d" will use eight characters to print the decimal value with spaces added to reach eight; "%08d" will use eight characters as well but will add zeroes instead of spaces.

The type of the arguments after *pcString* must match the requirements of the format string. For example, if an integer was passed where a string was expected, an error of some kind will most likely occur.

This function is contained in `utils/uartstdio.c`, with `utils/uartstdio.h` containing the API definition for use by applications.

Returns:

None.

27.2.2.44 UARTStdioInit

Initializes the UART console.

Prototype:

```
void
UARTStdioInit(unsigned long ulPortNum)
```

Parameters:

ulPortNum is the number of UART port to use for the serial console (0-2)

Description:

This function will initialize the specified serial port to be used as a serial console. The serial parameters will be set to 115200, 8-N-1.

This function must be called prior to using any of the other UART console functions: UARTprintf() or UARTgets(). In order for this function to work correctly, SysCtlClockSet() must be called prior to calling this function.

It is assumed that the caller has previously configured the relevant UART pins for operation as a UART rather than as GPIOs.

This function is contained in `utils/uartstdio.c`, with `utils/uartstdio.h` containing the API definition for use by applications.

Returns:

None.

27.2.2.45 ulocaltime

Converts from seconds to calendar date and time.

Prototype:

```
void
ulocaltime(unsigned long ulTime,
           tTime *psTime)
```

Parameters:

ulTime is the number of seconds.

psTime is a pointer to the time structure that is filled in with the broken down date and time.

Description:

This function converts a number of seconds since midnight GMT on January 1, 1970 (traditional Unix epoch) into the equivalent month, day, year, hours, minutes, and seconds representation.

Returns:

None.

27.2.2.46 usnprintf

A simple snprintf function supporting %c, %d, %p, %s, %u, %x, and %X.

Prototype:

```
int
usnprintf(char *pcBuf,
          unsigned long ulSize,
```

```
            const char *pcString,
            ...)
```

Parameters:

pcBuf is the buffer where the converted string is stored.

ulSize is the size of the buffer.

pcString is the format string.

... are the optional arguments, which depend on the contents of the format string.

Description:

This function is very similar to the C library `sprintf()` function. Only the following formatting characters are supported:

- %c to print a character
- %d to print a decimal value
- %s to print a string
- %u to print an unsigned decimal value
- %x to print a hexadecimal value using lower case letters
- %X to print a hexadecimal value using lower case letters (not upper case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %d, %p, %s, %u, %x, and %X, an optional number may reside between the % and the format character, which specifies the minimum number of characters to use for that value; if preceeded by a 0 then the extra characters will be filled with zeros instead of spaces. For example, "%8d" will use eight characters to print the decimal value with spaces added to reach eight; "%08d" will use eight characters as well but will add zeros instead of spaces.

The type of the arguments after *pcString* must match the requirements of the format string. For example, if an integer was passed where a string was expected, an error of some kind will most likely occur.

The function will copy at most *ulSize* - 1 characters into the buffer *pcBuf*. One space is reserved in the buffer for the null termination character.

The function will return the number of characters that would be converted as if there were no limit on the buffer size. Therefore it is possible for the function to return a count that is greater than the specified buffer size. If this happens, it means that the output was truncated.

Returns:

Returns the number of characters that were to be stored, not including the NULL termination character, regardless of space in the buffer.

27.2.2.47 usprintf

A simple sprintf function supporting %c, %d, %p, %s, %u, %x, and %X.

Prototype:

```
int
usprintf(char *pcBuf,
         const char *pcString,
         ...)
```

Parameters:

pcBuf is the buffer where the converted string is stored.

pcString is the format string.

... are the optional arguments, which depend on the contents of the format string.

Description:

This function is very similar to the C library `sprintf()` function. Only the following formatting characters are supported:

- %c to print a character
- %d to print a decimal value
- %s to print a string
- %u to print an unsigned decimal value
- %x to print a hexadecimal value using lower case letters
- %X to print a hexadecimal value using lower case letters (not upper case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %d, %p, %s, %u, %x, and %X, an optional number may reside between the % and the format character, which specifies the minimum number of characters to use for that value; if preceeded by a 0 then the extra characters will be filled with zeros instead of spaces. For example, “%8d” will use eight characters to print the decimal value with spaces added to reach eight; “%08d” will use eight characters as well but will add zeros instead of spaces.

The type of the arguments after *pcString* must match the requirements of the format string. For example, if an integer was passed where a string was expected, an error of some kind will most likely occur.

The caller must ensure that the buffer *pcBuf* is large enough to hold the entire converted string, including the null termination character.

Returns:

Returns the count of characters that were written to the output buffer, not including the NULL termination character.

27.2.2.48 ustrstr

Finds a substring within a string.

Prototype:

```
char *
ustrstr(const char *pcHaystack,
        const char *pcNeedle)
```

Parameters:

pcHaystack is a pointer to the string that will be searched.

pcNeedle is a pointer to the substring that is to be found within *pcHaystack*.

Description:

This function is very similar to the C library `strstr()` function. It scans a string for the first instance of a given substring and returns a pointer to that substring. If the substring cannot be found, a NULL pointer is returned.

Returns:
Returns a pointer to the first occurrence of *pcNeedle* within *pcHaystack* or NULL if no match is found.

27.2.2.49 ustrtoul

Converts a string into its numeric equivalent.

Prototype:
```
unsigned long
ustrtoul(const char *pcStr,
         const char **ppcStrRet,
         int iBase)
```

Parameters:
pcStr is a pointer to the string containing the integer.
ppcStrRet is a pointer that will be set to the first character past the integer in the string.
iBase is the radix to use for the conversion; can be zero to auto-select the radix or between 2 and 16 to explicitly specify the radix.

Description:
This function is very similar to the C library `strtoul()` function. It scans a string for the first token (that is, non-whitespace) and converts the value at that location in the string into an integer value.

Returns:
Returns the result of the conversion.

27.2.2.50 uvsnprintf

A simple vsnprintf function supporting %c, %d, %p, %s, %u, %x, and %X.

Prototype:
```
int
uvsnprintf(char *pcBuf,
           unsigned long ulSize,
           const char *pcString,
           va_list vaArgP)
```

Parameters:
pcBuf points to the buffer where the converted string is stored.
ulSize is the size of the buffer.
pcString is the format string.
vaArgP is the list of optional arguments, which depend on the contents of the format string.

Description:
This function is very similar to the C library `vsnprintf()` function. Only the following formatting characters are supported:

- %c to print a character

- %d to print a decimal value
- %s to print a string
- %u to print an unsigned decimal value
- %x to print a hexadecimal value using lower case letters
- %X to print a hexadecimal value using lower case letters (not upper case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %d, %p, %s, %u, %x, and %X, an optional number may reside between the % and the format character, which specifies the minimum number of characters to use for that value; if preceeded by a 0 then the extra characters will be filled with zeros instead of spaces. For example, “%8d” will use eight characters to print the decimal value with spaces added to reach eight; “%08d” will use eight characters as well but will add zeroes instead of spaces.

The type of the arguments after *pcString* must match the requirements of the format string. For example, if an integer was passed where a string was expected, an error of some kind will most likely occur.

The *ulSize* parameter limits the number of characters that will be stored in the buffer pointed to by *pcBuf* to prevent the possibility of a buffer overflow. The buffer size should be large enough to hold the expected converted output string, including the null termination character.

The function will return the number of characters that would be converted as if there were no limit on the buffer size. Therefore it is possible for the function to return a count that is greater than the specified buffer size. If this happens, it means that the output was truncated.

Returns:

Returns the number of characters that were to be stored, not including the NULL termination character, regardless of space in the buffer.

28 Error Handling

Invalid arguments and error conditions are handled in a non-traditional manner in the peripheral driver library. Typically, a function would check its arguments to make sure that they are valid (if required; some may be unconditionally valid such as a 32-bit value used as the load value for a 32-bit timer). If an invalid argument is provided, it would return an error code. The caller then has to check the return code from each invocation of the function to make sure that it succeeded.

This results in a sizable amount of argument checking code in each function and return code checking code at each call site. For a self-contained application, this extra code becomes an unneeded burden once the application is debugged. Having a means of removing it allows the final code to be smaller and therefore run faster.

In the peripheral driver library, most functions do not return errors (FlashProgram(), FlashErase(), FlashProtectSet(), and FlashProtectSave() are the notable exceptions). Argument checking is done via a call to the `ASSERT` macro (provided in `src/debug.h`). This macro has the usual definition of an assert macro; it takes an expression that "must" be true. By making this macro be empty, the argument checking is removed from the code.

There are two definitions of the `ASSERT` macro provided in `src/debug.h`; one that is empty (used for normal situations) and one that evaluates the expression (used when the library is built with debugging). The debug version will call the `__error__` function whenever the expression is not true, passing the file name and line number of the `ASSERT` macro invocation. The `__error__` function is prototyped in `src/debug.h` and must be provided by the application since it is the application's responsibility to deal with error conditions.

By setting a breakpoint on the `__error__` function, the debugger will immediately stop whenever an error occurs anywhere in the application (something that would be very difficult to do with other error checking methods). When the debugger stops, the arguments to the `__error__` function and the backtrace of the stack will pinpoint the function that found an error, what it found to be a problem, and where it was called from. As an example:

```
void
UARTParityModeSet(unsigned long ulBase, unsigned long ulParity)
{
    //
    // Check the arguments.
    //
    ASSERT((ulBase == UART0_BASE) || (ulBase == UART1_BASE) ||
           (ulBase == UART2_BASE));
    ASSERT((ulParity == UART_CONFIG_PAR_NONE) ||
           (ulParity == UART_CONFIG_PAR_EVEN) ||
           (ulParity == UART_CONFIG_PAR_ODD) ||
           (ulParity == UART_CONFIG_PAR_ONE) ||
           (ulParity == UART_CONFIG_PAR_ZERO));
```

Each argument is individually checked, so the line number of the failing `ASSERT` will indicate the argument that is invalid. The debugger will be able to display the values of the arguments (from the stack backtrace) as well as the caller of the function that had the argument error. This allows the problem to be quickly identified at the cost of a small amount of code.

29 Boot Loader

29.1 Introduction

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for applications running on a Stellaris microcontroller. The boot loader can be built to use either the UART0, SSI0, I2C0, or Ethernet ports to update the code on the microcontroller. The boot loader is customizable via source code modifications, or simply deciding at compile time which routines to include. Since full source code is provided, the boot loader can be completely customized.

Two update protocols are utilized. On UART0, SSI0, and I2C0, a custom protocol is used to communicate with the download utility to transfer the firmware image and program it into flash. When using Ethernet, the standard bootstrap protocol (BOOTP) is used instead.

When configured to use UART0 or Ethernet, the LM Flash Programmer GUI can be used to download an application via the boot loader. The LM Flash Programmer utility is available for download from www.luminarymicro.com.

Note:
Building the boot loader requires the use of linker scripts, and building applications that run under its control requires the ability to specify a start address other than the beginning of flash. Neither of these capabilities are available in the evaluation version of Keil RealView Microcontroller Development Kit; therefore, the boot loader is not usable unless the full version is used. Additionally, the linker script specified in the uVision project file for the boot loader is simply ignored, resulting in a successful link of the boot loader but an image that will not operate correctly.

29.1.1 Source Files

The following is an overview of the organization of the source code provided with the boot loader.

bl_autobaud.c	The code for performing the auto-baud operation on the UART port. This is separate from the remainder of the UART code so that the linker can remove it when it is not used.
bl_check.c	The code to check if a firmware update is required, or if a firmware update is being requested by the user.
bl_check.h	Prototypes for the update check code.
bl_commands.h	The list of commands and return messages supported by the boot loader.

bl_config.c	A dummy source file used to translate the bl_config.h C header file into a header file that can be included in assembly code. This is needed for the Keil tool chain since it is not able to pass assembly source code through the C preprocessor.
bl_config.h.tmpl	A template for the boot loader configuration file. This contains all of the possible configuration values.
bl_decrypt.c	The code to perform an in-place decryption of the downloaded firmware image. No decryption is actually performed in this file; this is simply a stub that can be expanded to perform the require decryption.
bl_decrypt.h	Prototypes for the in-place decryption routines.
bl_enet.c	The functions for performing a firmware update via the Ethernet port.
bl_i2c.c	The functions for transferring data via the I2C0 port.
bl_i2c.h	Prototypes for the I2C0 transfer functions.
bl_link.ld	The linker script used when the codered, gcc, or sourcerygxx compiler is being used to build the boot loader.
bl_link.sct	The linker script used when the rvmdk compiler is being used to build the boot loader.
bl_link.xcl	The linker script used when the ewarm compiler is being used to build the boot loader.
bl_main.c	The main control loop of the boot loader.
bl_packet.c	The functions for handling the packet processing of commands and responses.
bl_packet.h	Prototypes for the packet handling functions.
bl_ssi.c	The functions for transferring data via the SSI0 port.
bl_ssi.h	Prototypes for the SSI0 transfer functions.
bl_startup_codered.S	The start-up code used when the codered compiler is being used to build the boot loader.
bl_startup_ewarm.S	The start-up code used when the ewarm compiler is being used to build the boot loader.
bl_startup_gcc.S	The start-up code used when the gcc compiler is being used to build the boot loader.

bl_startup_rvmdk.S	The start-up code used when the rvmdk compiler is being used to build the boot loader.
bl_startup_sourcerygxx.S	The start-up code used when the sourcerygxx compiler is being used to build the boot loader.
bl_uart.c	The functions for transferring data via the UART0 port.
bl_uart.h	Prototypes for the UART0 transfer functions.

29.1.2 Start-up Code

The start-up code contains the minimal set of code required to configure a vector table, initialize memory, copy the boot loader from flash to SRAM, and execute from SRAM. Because some tool chain-specific constructs are used to indicate where the code, data, and bss segments reside in memory, each supported tool chain has its own separate file that implements the start-up code. The start-up code is contained in the following files:

- bl_startup_codered.S (Code Red Technologies tools)
- bl_startup_ewarm.S (IAR Embedded Workbench)
- bl_startup_gcc.S (GNU GCC)
- bl_startup_rvmdk.S (Keil RV-MDK)
- bl_startup_sourcerygxx.S (CodeSourcery Sourcery G++)

Accompanying the start-up code for each tool chain are linker scripts that are used to place the vector table, code segment, data segment initializers, and data segments in the appropriate locations in memory. The scripts are located in the following files:

- bl_link.ld (Code Red Technologies tools, GNU GCC, and CodeSourcery Sourcery G++)
- bl_link.sct (Keil RV-MDK)
- bl_link.xcl (IAR Embedded Workbench)

The boot loader's code and its corresponding linker script use a memory layout that exists entirely in SRAM. This means that the load address of the code and read-only data are not the same as the execution address. This memory map allows the boot loader to update itself since it is actually running from SRAM only. The first part of SRAM is used as the copy space for the boot loader while the rest is reserved for stack and read/write data for the boot loader. Once the boot loader calls the application, all SRAM becomes usable by the application.

The vector table of the Cortex-M3 microprocessor contains four required entries: the initial stack pointer, the reset handler address, the NMI handler address, and the hard fault handler address. Upon reset, the processor loads the initial stack pointer and then starts executing the reset handler. The initial stack pointer is required since an NMI or hard fault can occur at any time; the stack is required to take those interrupts since the processor automatically pushes eight items onto the stack.

The Vectors array contains the boot loader's vector table which varies in size based on the addition of the auto-baud feature. The auto-baud feature requires an interrupt and expands the vector

table slightly. Since the boot loader executes from SRAM and not from flash, tool chain-specific constructs are used to provide a hint to the linker that this array is located at 0x2000.0000.

The IntDefaultHandler function contains the default fault handler. This is a simple infinite loop, effectively halting the application if any unexpected fault occurs. The application state is, therefore, preserved for examination by a debugger. If desired, a customized boot loader can provide its own handlers by adding the appropriate handlers to the Vectors array.

After a reset, the start-up copies the boot loader from flash to SRAM, branches to the copy of the boot loader in SRAM, and checks to see if an application update should be performed by calling CheckForceUpdate(). If an update is not required, the application is called. Otherwise, the microcontroller is initialized by calling ConfigureDevice() (for UART0, SSI0, and I2C0) or ConfigureEnet() (for Ethernet), and then the control loop of the boot loader in Updater() (for UART0, SSI0, and I2C0) or UpdateBOOTP() is called.

The check for an application update (in CheckForceUpdate()) consists of checking the beginning of the application area and optionally checking the state of a GPIO pin. The application is assumed to be valid if the first location is a valid stack pointer (that is, it resides in SRAM, and has a value of 0x2xxx.xxxx), and the second location is a valid reset handler address (that is, it resides in flash, and has a value of 0x000x.xxxx, where the value is odd). If either of these tests fail, then the application is assumed to be invalid and an update is forced. The GPIO pin check can be enabled with **ENABLE_UPDATE_CHECK**, in which case an update can be forced by changing the state of a GPIO pin (for example, with a push button). If the application is valid and the GPIO pin is not requesting an update, the application is called. Otherwise, an update is started by entering the main loop of the boot loader.

Additionally, the boot loader can be called by the application in order to perform an application-directed update. In this case, the boot loader assumes that the peripheral in use for the update has already been configured by the application, and must simply be used by the boot loader to perform the update. The boot loader therefore copies itself to SRAM, branches to the SRAM copy of the boot loader, and starts the update by calling Updater() (for UART0, SSI0, and I2C0) or UpdateBOOTP() (for Ethernet). The SVCall entry of the vector table contains the location of the application-directed update entry point.

29.1.3 Ethernet Update

When performing an Ethernet update, ConfigureEnet() is used to configure the Ethernet controller, making it ready to be used to update the firmware. Then, UpdateBOOTP() begins the process of the firmware update.

The bootstrap protocol (BOOTP) is a predecessor to the DHCP protocol and is used to discover the IP address of the client, the IP address of the server, and the name of the firmware image to use. BOOTP uses UDP/IP packets to communicate between the client and the server; the boot loader acts as the client. First, it will send a BOOTP request using a broadcast message. When the server receives the request, it will reply, thereby informing the client of its IP address, the IP address of the server, and the name of the firmware image. Once this reply is received, the BOOTP protocol has completed.

Then, the trivial file transfer protocol (TFTP) is used to transfer the firmware image from the server to the client. TFTP also uses UDP/IP packets to communicate between the client and the server, and the boot loader also acts as the client in this protocol. As each data block is received, it is programmed into flash. Once all data blocks are received and programmed, the device is reset, causing it to start running the new firmware image.

The uIP stack (http://www.sics.se/~adam/uip) is used to implement the UDP/IP connections. The TCP support is not needed and is therefore disabled, greatly reducing the size of the stack.

Note:
When using the Ethernet update, the boot loader can not update itself since there is no mechanism in BOOTP to distinguish between a firmware image and a boot loader image. Therefore, the boot loader does not know if a given image is a new boot loader or a new firmware image. It assumes that all images provided are firmware images.

- RFC951 (http://tools.ietf.org/html/rfc951.html) defines the bootstrap protocol.
- RFC1350 (http://tools.ietf.org/html/rfc1350.html) defines the trivial file transfer protocol.

29.1.4 Serial Update

When performing an update via a serial port (UART0, SSI0, or I2C0), `ConfigureDevice()` is used to configure the selected serial port, making it ready to be used to update the firmware. Then, `Updater()` sits in an endless loop accepting commands and updating the firmware when requested. The commands are explained in detail in the Commands section. All transmissions from this main routine use the packet handler functions (`SendPacket()`, `ReceivePacket()`, `AckPacket()`, and `NakPacket()`). Once the update is complete, the boot loader can be reset by issuing a reset command to the boot loader.

When a request to update the application comes through and **FLASH_CODE_PROTECTION** is defined, the boot loader first erases the entire application area before accepting the binary for the new application. This prevents a partial erase of flash from exposing any of the code before the new binary is downloaded to the microcontroller. The boot loader itself is left in place so that it will not boot a partially erased program. Once all of the application flash area has been successfully erased, the boot loader proceeds with the download of the new binary. When **FLASH_CODE_PROTECTION** is not defined, the boot loader only erases enough space to fit the new application that is being downloaded.

In the event that the boot loader itself needs to be updated, the boot loader must replace itself in flash. An update of the boot loader is recognized by performing a download to address 0x0000.0000. Once again the boot loader operates differently based on the setting of **FLASH_CODE_PROTECTION**. When **FLASH_CODE_PROTECTION** is defined and the download address indicates an boot loader update, the boot loader protects any application code already on the microcontroller by erasing the entire application area before erasing and replacing itself. If the process is interrupted at any point, either the old boot loader remains present in the flash and does not boot the partial application or the application code will have already been erased. When **FLASH_CODE_PROTECTION** is not defined, the boot loader only erases enough space to fit its own code and leaves the application intact.

29.1.4.1 Packet Handling

The boot loader uses well-defined packets to ensure reliable communications with the update program. The packets are always acknowledged or not acknowledged by the communicating devices. The packets use the same format for receiving and sending packets. This includes the method used to acknowledge successful or unsuccessful reception of a packet. While the actual signaling

on the serial ports is different, the packet format remains independent of the method of transporting the data.

The boot loader uses the `SendPacket()` function in order to send a packet of data to another device. This function encapsulates all of the steps necessary to send a valid packet to another device including waiting for the acknowledge or not-acknowledge from the other device. The following steps must be performed to successfully send a packet:

1. Send out the size of the packet that will be sent to the device. The size is always the size of the data + 2.
2. Send out the checksum of the data buffer to help ensure proper transmission of the command. The checksum algorithm is implemented in the `CheckSum()` function provided and is simply a sum of the data bytes.
3. Send out the actual data bytes.
4. Wait for a single byte acknowledgment from the device that it either properly received the data or that it detected an error in the transmission.

Received packets use the same format as sent packets. The boot loader uses the `ReceivePacket()` function in order to receive or wait for a packet from another device. This function does not take care of acknowledging or not-acknowledging the packet to the other device. This allows the contents of the packet to be checked before sending back a response. The following steps must be performed to successfully receive a packet:

1. Wait for non-zero data to be returned from the device. This is important as the device may send zero bytes between a sent and received data packet. The first non-zero byte received will be the size of the packet that is being received.
2. Read the next byte which will be the checksum for the packet.
3. Read the data bytes from the device. There will be packet size - 2 bytes of data sent during the data phase. For example, if the packet size was 3, then there is only 1 byte of data to be received.
4. Calculate the checksum of the data bytes and ensure if it matches the checksum received in the packet.
5. Send an acknowledge or not-acknowledge to the device to indicate the successful or unsuccessful reception of the packet.

The steps necessary to acknowledge reception of a packet are implemented in the `AckPacket()` function. Acknowledge bytes are sent out whenever a packet is successfully received and verified by the boot loader.

A not-acknowledge byte is sent out whenever a sent packet is detected to have an error, usually as a result of a checksum error or just malformed data in the packet. This allows the sender to re-transmit the previous packet.

29.1.4.2 Transport Layer

The boot loader supports updating via the I2C0, SSI0, and UART0 ports which are available on Stellaris microcontrollers. The SSI port has the advantage of supporting higher and more flexible data rates but it also requires more connections to the microcontroller. The UART has the disadvantage of having slightly lower and possibly less flexible rates. However, the UART requires fewer pins and can be easily implemented with any standard UART connection. The I2C interface also provides a standard interface, only uses two wires, and can operate at comparable speeds to the UART and SSI interfaces.

29.1.4.3 I2C Transport

The I2C handling functions are `I2CSend()`, `I2CReceive()`, and `I2CFlush()` functions. The connections required to use the I2C port are the following pins: **I2CSCL** and **I2CSDA**. The device communicating with the boot loader must operate as the I2C master and provide the **I2CSCL** signal. The **I2CSDA** pin is open drain and can be driven by either the master or the slave I2C device.

29.1.4.4 SSI Transport

The SSI handling functions are `SSISend()`, `SSIReceive()`, and `SSIFlush()`. The connections required to use the SSI port are the following four pins: **SSITx**, **SSIRx**, **SSIClk**, and **SSIFss**. The device communicating with the boot loader is responsible for driving the **SSIRx**, **SSIClk**, and **SSIFss** pins, while the Stellaris microcontroller drives the **SSITx** pin. The format used for SSI communications is the Motorola format with SPH set to 1 and SPO set to 1 (see Stellaris Family data sheet for more information on this format). The SSI interface has a hardware requirement that limits the maximum rate of the SSI clock to be at most 1/12 the frequency of the microcontroller running the boot loader.

29.1.4.5 UART Transport

The UART handling functions are `UARTSend()`, `UARTReceive()`, and `UARTFlush()`. The connections required to use the UART port are the following two pins: **U0Tx** and **U0Rx**. The device communicating with the boot loader is responsible for driving the **U0Rx** pin on the Stellaris microcontroller, while the Stellaris microcontroller drives the **U0Tx** pin.

While the baud rate is flexible, the UART serial format is fixed at 8 data bits, no parity, and one stop bit. The baud rate used for communication can either be auto-detected by the boot loader, if the auto-baud feature is enabled, or it can be fixed at a baud rate supported by the device communicating with the boot loader. The only requirement on baud rate is that the baud rate should be no more than 1/32 the frequency of the microcontroller that is running the boot loader. This is the hardware requirement for the maximum baud rate for a UART on any Stellaris microcontroller.

When using a fixed baud rate, the frequency of the crystal connected to the microcontroller must be specified. Otherwise, the boot loader will not be able to configure the UART to operate at the requested baud rate.

The boot loader provides a method to automatically detect the baud rate being used to communicate with it. This automatic baud rate detection is implemented in the `UARTAutoBaud()` function. The auto-baud function attempts to synchronize with the updater application and indicates if it is successful in detecting the baud rate or if it failed to properly detect the baud rate. The boot loader can make multiple calls to `UARTAutoBaud()` to attempt to retry the synchronization if the first call fails. In the example boot loader provided, when the auto-baud feature is enabled, the boot loader will wait forever for a valid synchronization pattern from the host.

29.1.5 Customization

The boot loader allows for customization of its features as well as the interfaces used to update the microcontroller. This allows the boot loader to include only the features that are needed by the application. There are two types of features that can be customized. The first type are the features

that are conditionally included or excluded at compile time. The second type of customizations are more involved and include customizing the actual code that is used by the boot loader.

The boot loader can be modified to have any functionality. As an example, the main loop can be completely replaced to use a different set of commands and still use the packet and transport functions from the boot loader. The method of detecting a forced update can be modified to suit the needs of the application when toggling a GPIO to detect an update request may not be the best solution. If the boot loader's packet format does not meet the needs of the application, it can be completely replaced by replacing `ReceivePacket()`, `SendPacket()`, `AckPacket()`, and `NakPacket()`.

The boot loader also provides a method to add a new transmission interfaces beyond the UART, SSI, and I2C that are provided by the boot loader. In order for the packet functions to use the new transport functions, the `SendData`, `ReceiveData`, and `FlushData` defines need to be modified to use the new functions. For example:

```
#ifdef FOO_ENABLE_UPDATE
#define SendData                FooSend
#define FlushData               FooFlush
#define ReceiveData             FooReceive
#endif
```

would use the functions for the hypothetical Foo peripheral.

The combination of these customizable features provides a framework that allows the boot loader to define whatever protocol it needs, or use any port that it needs to perform updates of the microcontroller.

29.1.6 Commands

The following commands are used by the custom protocol on the UART0, SSI0, and I2C0 ports:

`COMMAND_PING`

This command is used to receive an acknowledge from the boot loader indicating that communication has been established. This command is a single byte.

The format of the command is as follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_PING;
```

COMMAND_DOWNLOAD

This command is sent to the boot loader to indicate where to store data and how many bytes will be sent by the COMMAND_SEND_DATA commands that follow. The command consists of two 32-bit values that are both transferred MSB first. The first 32-bit value is the address to start programming data into, while the second is the 32-bit size of the data that will be sent. This command also triggers an erasure of the full application area in the flash or possibly the entire flash depending on the address used. This causes the command to take longer to send the ACK/NAK in response to the command. This command should be followed by a COMMAND_GET_STATUS to ensure that the program address and program size were valid for the microcontroller running the boot loader.

The format of the command is as follows:

```
unsigned char ucCommand[9];

ucCommand[0] = COMMAND_DOWNLOAD;
ucCommand[1] = Program Address [31:24];
ucCommand[2] = Program Address [23:16];
ucCommand[3] = Program Address [15:8];
ucCommand[4] = Program Address [7:0];
ucCommand[5] = Program Size [31:24];
ucCommand[6] = Program Size [23:16];
ucCommand[7] = Program Size [15:8];
ucCommand[8] = Program Size [7:0];
```

COMMAND_RUN

This command is sent to the boot loader to transfer execution control to the specified address. The command is followed by a 32-bit value, transferred MSB first, that is the address to which execution control is transferred.

The format of the command is as follows:

```
unsigned char ucCommand[5];

ucCommand[0] = COMMAND_RUN;
ucCommand[1] = Run Address [31:24];
ucCommand[2] = Run Address [23:16];
ucCommand[3] = Run Address [15:8];
ucCommand[4] = Run Address [7:0];
```

COMMAND_GET_STATUS

This command returns the status of the last command that was issued. Typically, this command should be received after every command is sent to ensure that the previous command was successful or, if unsuccessful, to properly respond to a failure. The command requires one byte in the data of the packet and the boot loader should respond by sending a packet with one byte of data that contains the current status code.

The format of the command is as follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_GET_STATUS;
```

The following are the definitions for the possible status values that can be returned from the boot loader when COMMAND_GET_STATUS is sent to the the microcontroller.

```
COMMAND_RET_SUCCESS
COMMAND_RET_UNKNOWN_CMD
COMMAND_RET_INVALID_CMD
COMMAND_RET_INVALID_ADD
COMMAND_RET_FLASH_FAIL
```

COMMAND_SEND_DATA

This command should only follow a COMMAND_DOWNLOAD command or another COMMAND_SEND_DATA command, if more data is needed. Consecutive send data commands automatically increment the address and continue programming from the previous location. The transfer size is limited by the size of the receive buffer in the boot loader (as configured by the BUFFER_SIZE parameter). The command terminates programming once the number of bytes indicated by the COMMAND_DOWNLOAD command has been received. Each time this function is called, it should be followed by a COMMAND_GET_STATUS command to ensure that the data was successfully programmed into the flash. If the boot loader sends a NAK to this command, the boot loader will not increment the current address which allows for retransmission of the previous data.

The format of the command is as follows:

```
unsigned char ucCommand[9];

ucCommand[0] = COMMAND_SEND_DATA
ucCommand[1] = Data[0];
ucCommand[2] = Data[1];
ucCommand[3] = Data[2];
ucCommand[4] = Data[3];
ucCommand[5] = Data[4];
ucCommand[6] = Data[5];
ucCommand[7] = Data[6];
ucCommand[8] = Data[7];
```

COMMAND_RESET

This command is used to tell the boot loader to reset. This is used after downloading a new image to the microcontroller to cause the new application or the new boot loader to start from a reset. The normal boot sequence occurs and the image runs as if from a hardware reset. It can also be used to reset the boot loader if a critical error occurs and the host device wants to restart communication with the boot loader.

The boot loader responds with an ACK signal to the host device before actually executing the software reset on the microcontroller running the boot loader. This informs the updater application that the command was received successfully and the part will be reset.

The format of the command is as follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_RESET;
```

29.1.7 Configuration

There are a number of defines that are used to configure the operation of the boot loader. These defines are located in the bl_config.h header file, for which there is a template (bl_config.h.tmpl) provided with the boot loader.

The configuration options are:

CRYSTAL_FREQ

This defines the crystal frequency used by the microcontroller running the boot loader. If this is unknown at the time of production, then use the UART_AUTOBAUD feature to properly configure the UART.

- This value must be defined if using the UART for the update and not using the auto-baud feature, and when using Ethernet for the update.

BOOST_LDO_VOLTAGE

This enables the boosting of the LDO voltage to 2.75V. For boot loader configurations that enable the PLL (in other words, using the Ethernet port) on a part that has the PLL errata, this should be enabled. This applies to revision A2 of Fury-class devices.

APP_START_ADDRESS	The starting address of the application. This must be a multiple of 1024 bytes (making it aligned to a page boundary). A vector table is expected at this location, and the perceived validity of the vector table (stack located in SRAM, reset vector located in flash) is used as an indication of the validity of the application image. ■ This value must be defined. The flash image of the boot loader must not be larger than this value.
FLASH_RSVD_SPACE	The amount of space at the end of flash to reserve. This must be a multiple of 1024 bytes (making it aligned to a page boundary). This reserved space is not erased when the application is updated, providing non-volatile storage that can be used for parameters.
STACK_SIZE	The number of words of stack space to reserve for the boot loader. ■ This value must be defined.
BUFFER_SIZE	The number of words in the data buffer used for receiving packets. This value must be at least 3. If using auto-baud on the UART, this must be at least 20. The maximum usable value is 65 (larger values will result in unused space in the buffer). This value is unused when updating via the Ethernet port. ■ This value must be defined.
ENABLE_BL_UPDATE	Enables updates to the boot loader. Updating the boot loader is an unsafe operation since it is not fully fault tolerant (losing power to the device partway through could result in the boot loader no longer being present in flash). The boot loader can not be updated via the Ethernet port.
FLASH_CODE_PROTECTION	This definition will cause the the boot loader to erase the entire flash on updates to the boot loader or to erase the entire application area when the application is updated. This erases any unused sections in the flash before the firmware is updated.
ENABLE_DECRYPTION	Enables the call to decrypt the downloaded data before writing it into flash. The decryption routine is empty in the reference boot loader source, which simply provides a placeholder for adding an actual decryption algorithm.
ENABLE_UPDATE_CHECK	Enables the pin-based forced update check. When enabled, the boot loader will go into update mode instead of calling the application if a pin is read at a particular polarity, forcing an update operation. In either case, the application is still able to return control to the boot loader in order to start an update.

FORCED_UPDATE_PERIPH	The GPIO module to enable in order to check for a forced update. This will be one of the SYSCTL_RCGC2_GPIOx values, where "x" is replaced with the port name (such as B). The value of "x" should match the value of "x" for FORCED_UPDATE_PORT. ■ This value must be defined if ENABLE_UPDATE_CHECK is defined.
FORCED_UPDATE_PORT	The GPIO port to check for a forced update. This will be one of the GPIO_PORTx_BASE values, where "x" is replaced with the port name (such as B). The value of "x" should match the value of "x" for FORCED_UPDATE_PERIPH. ■ This value must be defined if ENABLE_UPDATE_CHECK is defined.
FORCED_UPDATE_PIN	The pin to check for a forced update. This is a value between 0 and 7. ■ This value must be defined if ENABLE_UPDATE_CHECK is defined.
FORCED_UPDATE_POLARITY	The polarity of the GPIO pin that results in a forced update. This value should be 0 if the pin should be low and 1 if the pin should be high. ■ This value must be defined if ENABLE_UPDATE_CHECK is defined.
UART_ENABLE_UPDATE	Selects the UART as the port for communicating with the boot loader.
UART_AUTOBAUD	Enables automatic baud rate detection. This can be used if the crystal frequency is unknown, or if operation at different baud rates is desired.
UART_FIXED_BAUDRATE	Selects the baud rate to be used for the UART.
SSI_ENABLE_UPDATE	Selects the SSI port as the port for communicating with the boot loader.
I2C_ENABLE_UPDATE	Selects the I2C port as the port for communicating with the boot loader.
I2C_SLAVE_ADDR	Specifies the I2C address of the boot loader. ■ This value must be defined if I2C_ENABLE_UPDATE is defined.
ENET_ENABLE_UPDATE	Selects an update via the Ethernet port.

ENET_ENABLE_LEDS	Enables the use of the Ethernet status LED outputs to indicate traffic and connection status.
ENET_MAC_ADDR?	Specifies the hard coded MAC address for the Ethernet interface. There are six individual values (ENET_MAC_ADDR0 through ENET_MAC_ADDR5) that provide the six bytes of the MAC address, where ENET_MAC_ADDR0 though ENET_MAC_ADDR2 provide the organizationally unique identifier (OUI) and ENET_MAC_ADDR3 through ENET_MAC_ADDR5 provide the extension identifier. If these values are not provided, the MAC address will be extracted from the user registers.
ENET_BOOTP_SERVER	Specifies the name of the BOOTP server from which to request information. The use of this specifier allows a board-specific BOOTP server to co-exist on a network with the DHCP server that may be part of the network infrastructure. The BOOTP server provided by Luminary Micro requires that this be set to "stellaris".

29.2 Functions

Functions

- void AckPacket (void)
- char BOOTPThread (void)
- unsigned long CheckForceUpdate (void)
- unsigned long CheckSum (const unsigned char *pucData, unsigned long ulSize)
- void ConfigureDevice (void)
- void ConfigureEnet (void)
- void DecryptData (unsigned char *pucBuffer, unsigned long ulSize)
- void GPIOIntHandler (void)
- void I2CFlush (void)
- void I2CReceive (unsigned char *pucData, unsigned long ulSize)
- void I2CSend (const unsigned char *pucData, unsigned long ulSize)
- void NakPacket (void)
- int ReceivePacket (unsigned char *pucData, unsigned long *pulSize)
- int SendPacket (unsigned char *pucData, unsigned long ulSize)
- void SSIFlush (void)
- void SSIReceive (unsigned char *pucData, unsigned long ulSize)
- void SSISend (const unsigned char *pucData, unsigned long ulSize)
- void SysTickIntHandler (void)
- int UARTAutoBaud (unsigned long *pulRatio)
- void UARTFlush (void)
- void UARTReceive (unsigned char *pucData, unsigned long ulSize)

- void UARTSend (const unsigned char *pucData, unsigned long ulSize)
- void UpdateBOOTP (void)
- void Updater (void)

29.2.1 Detailed Description

The following functions make up the boot loader. In order to keep the size of the boot loader to a minimum, none of the APIs from the peripheral driver library are utilized.

29.2.2 Function Documentation

29.2.2.1 AckPacket

Sends an Acknowledge packet.

Prototype:

```
void
AckPacket(void)
```

Description:

This function is called to acknowledge that a packet has been received by the microcontroller.

This function is contained in `bl_packet.c`.

Returns:

None.

29.2.2.2 BOOTPThread

Handles the BOOTP process.

Prototype:

```
char
BOOTPThread(void)
```

Description:

This function contains the proto-thread for handling the BOOTP process. It first communicates with the BOOTP server to get its boot parameters (IP address, server address, and filename), then it communicates with the TFTP server on the specified server to read the firmware image file.

This function is contained in `bl_enet.c`.

Returns:

None.

29.2.2.3 CheckForceUpdate

Checks if an update is needed or is being requested.

Prototype:

```
unsigned long
CheckForceUpdate(void)
```

Description:

This function detects if an update is being requested or if there is no valid code presently located on the microcontroller. This is used to tell whether or not to enter update mode.

This function is contained in `bl_check.c`.

Returns:

Returns a non-zero value if an update is needed or is being requested and zero otherwise.

29.2.2.4 CheckSum

Calculates an 8-bit checksum

Prototype:

```
unsigned long
CheckSum(const unsigned char *pucData,
         unsigned long ulSize)
```

Parameters:

pucData is a pointer to an array of 8-bit data of size ulSize.

ulSize is the size of the array that will run through the checksum algorithm.

Description:

This function simply calculates an 8-bit checksum on the data passed in.

This function is contained in `bl_packet.c`.

Returns:

Returns the calculated checksum.

29.2.2.5 ConfigureDevice

Configures the microcontroller.

Prototype:

```
void
ConfigureDevice(void)
```

Description:

This function configures the peripherals and GPIOs of the microcontroller, preparing it for use by the boot loader. The interface that has been selected as the update port will be configured, and auto-baud will be performed if required.

This function is contained in `bl_main.c`.

Returns:
None.

29.2.2.6 ConfigureEnet

Configures the Ethernet controller.

Prototype:
```
void
ConfigureEnet(void)
```

Description:
This function configures the Ethernet controller, preparing it for use by the boot loader.

This function is contained in `bl_enet.c`.

Returns:
None.

29.2.2.7 DecryptData

Performs an in-place decryption of downloaded data.

Prototype:
```
void
DecryptData(unsigned char *pucBuffer,
            unsigned long ulSize)
```

Parameters:
pucBuffer is the buffer that holds the data to decrypt.
ulSize is the size, in bytes, of the buffer that was passed in via the pucBuffer parameter.

Description:
This function is a stub that could provide in-place decryption of the data that is being downloaded to the device.

This function is contained in `bl_decrypt.c`.

Returns:
None.

29.2.2.8 GPIOIntHandler

Handles the UART Rx GPIO interrupt.

Prototype:
```
void
GPIOIntHandler(void)
```

Description:

When an edge is detected on the UART Rx pin, this function is called to save the time of the edge. These times are later used to determine the ratio of the UART baud rate to the processor clock rate.

This function is contained in `bl_autobaud.c`.

Returns:

None.

29.2.2.9 I2CFlush

Waits until all data has been transmitted by the I2C port.

Prototype:

```
void
I2CFlush(void)
```

Description:

This function waits until all data written to the I2C port has been read by the master.

This function is contained in `bl_i2c.c`.

Returns:

None.

29.2.2.10 I2CReceive

Receives data over the I2C port.

Prototype:

```
void
I2CReceive(unsigned char *pucData,
           unsigned long ulSize)
```

Parameters:

pucData is the buffer to read data into from the I2C port.

ulSize is the number of bytes provided in the *pucData* buffer that should be written with data from the I2C port.

Description:

This function reads back *ulSize* bytes of data from the I2C port, into the buffer that is pointed to by *pucData*. This function will not return until *ulSize* number of bytes have been received. This function will wait till the I2C Slave port has been properly addressed by the I2C Master before reading the first byte of data from the I2C port.

This function is contained in `bl_i2c.c`.

Returns:

None.

29.2.2.11 I2CSend

Sends data over the I2C port.

Prototype:

```
void
I2CSend(const unsigned char *pucData,
        unsigned long ulSize)
```

Parameters:

pucData is the buffer containing the data to write out to the I2C port.

ulSize is the number of bytes provided in *pucData* buffer that will be written out to the I2C port.

Description:

This function sends *ulSize* bytes of data from the buffer pointed to by *pucData* via the I2C port. The function will wait till the I2C Slave port has been properly addressed by the I2C Master device before sending the first byte.

This function is contained in `bl_i2c.c`.

Returns:

None.

29.2.2.12 NakPacket

Sends a no-acknowledge packet.

Prototype:

```
void
NakPacket(void)
```

Description:

This function is called when an invalid packet has been received by the microcontroller, indicating that it should be retransmitted.

This function is contained in `bl_packet.c`.

Returns:

None.

29.2.2.13 ReceivePacket

Receives a data packet.

Prototype:

```
int
ReceivePacket(unsigned char *pucData,
              unsigned long *pulSize)
```

Parameters:

pucData is the location to store the data that is sent to the boot loader.

pulSize is the number of bytes returned in the pucData buffer that was provided.

Description:
This function receives a packet of data from specified transfer function.

This function is contained in `bl_packet.c`.

Returns:
Returns zero to indicate success while any non-zero value indicates a failure.

29.2.2.14 SendPacket

Sends a data packet.

Prototype:
```
int
SendPacket(unsigned char *pucData,
           unsigned long ulSize)
```

Parameters:
pucData is the location of the data to be sent.
ulSize is the number of bytes to send.

Description:
This function sends the data provided in the *pucData* parameter in the packet format used by the boot loader. The caller only needs to specify the buffer with the data that needs to be transferred. This function addresses all other packet formatting issues.

This function is contained in `bl_packet.c`.

Returns:
Returns zero to indicate success while any non-zero value indicates a failure.

29.2.2.15 SSIFlush

Waits until all data has been transmitted by the SSI port.

Prototype:
```
void
SSIFlush(void)
```

Description:
This function waits until all data written to the SSI port has been read by the master.

This function is contained in `bl_ssi.c`.

Returns:
None.

29.2.2.16 SSIReceive

Receives data from the SSI port in slave mode.

Prototype:

```
void
SSIReceive(unsigned char *pucData,
           unsigned long ulSize)
```

Parameters:

pucData is the location to store the data received from the SSI port.

ulSize is the number of bytes of data to receive.

Description:

This function receives data from the SSI port in slave mode. The function will not return until *ulSize* number of bytes have been received.

This function is contained in `bl_ssi.c`.

Returns:

None.

29.2.2.17 SSISend

Sends data via the SSI port in slave mode.

Prototype:

```
void
SSISend(const unsigned char *pucData,
        unsigned long ulSize)
```

Parameters:

pucData is the location of the data to send through the SSI port.

ulSize is the number of bytes of data to send.

Description:

This function sends data through the SSI port in slave mode. This function will not return until all bytes are sent.

This function is contained in `bl_ssi.c`.

Returns:

None.

29.2.2.18 SysTickIntHandler

Handles the SysTick interrupt.

Prototype:

```
void
SysTickIntHandler(void)
```

Description:
This function is called when the SysTick interrupt occurs. It simply keeps a running count of interrupts, used as a time basis for the BOOTP and TFTP protocols.

This function is contained in `bl_enet.c`.

Returns:
None.

29.2.2.19 UARTAutoBaud

Performs auto-baud on the UART port.

Prototype:
```
int
UARTAutoBaud(unsigned long *pulRatio)
```

Parameters:
pulRatio is the ratio of the processor's crystal frequency to the baud rate being used by the UART port for communications.

Description:
This function attempts to synchronize to the updater program that is trying to communicate with the boot loader. The UART port is monitored for edges using interrupts. Once enough edges are detected, the boot loader determines the ratio of baud rate and crystal frequency needed to program the UART.

This function is contained in `bl_autobaud.c`.

Returns:
Returns a value of 0 to indicate that this call successfully synchronized with the other device communicating over the UART, and a negative value to indicate that this function did not successfully synchronize with the other UART device.

29.2.2.20 UARTFlush

Waits until all data has been transmitted by the UART port.

Prototype:
```
void
UARTFlush(void)
```

Description:
This function waits until all data written to the UART port has been transmitted.

This function is contained in `bl_uart.c`.

Returns:
None.

29.2.2.21 UARTReceive

Receives data over the UART port.

Prototype:

```
void
UARTReceive(unsigned char *pucData,
            unsigned long ulSize)
```

Parameters:

pucData is the buffer to read data into from the UART port.

ulSize is the number of bytes provided in the *pucData* buffer that should be written with data from the UART port.

Description:

This function reads back *ulSize* bytes of data from the UART port, into the buffer that is pointed to by *pucData*. This function will not return until *ulSize* number of bytes have been received.

This function is contained in `bl_uart.c`.

Returns:

None.

29.2.2.22 UARTSend

Sends data over the UART port.

Prototype:

```
void
UARTSend(const unsigned char *pucData,
         unsigned long ulSize)
```

Parameters:

pucData is the buffer containing the data to write out to the UART port.

ulSize is the number of bytes provided in *pucData* buffer that will be written out to the UART port.

Description:

This function sends *ulSize* bytes of data from the buffer pointed to by *pucData* via the UART port.

This function is contained in `bl_uart.c`.

Returns:

None.

29.2.2.23 UpdateBOOTP

Starts the update process via BOOTP.

Prototype:

```
void
UpdateBOOTP(void)
```

Description:

This function starts the Ethernet firmware update process. The BOOTP (as defined by RFC951 at http://tools.ietf.org/html/rfc951) and TFTP (as defined by RFC1350 at http://tools.ietf.org/html/rfc1350) protocols are used to transfer the firmware image over Ethernet.

This function is contained in `bl_enet.c`.

Returns:

Never returns.

29.2.2.24 Updater

This function performs the update on the selected port.

Prototype:

```
void
Updater(void)
```

Description:

This function is called directly by the boot loader or it is called as a result of an update request from the application.

This function is contained in `bl_main.c`.

Returns:

Never returns.

30 Tool Chain Specifics

30.1 Introduction

There are two aspects to the library's interaction with the supported tool chains; how it is built by the compilers and how it interacts with the debuggers. By separating it in this way, it is possible to use one tool chain to build the code and use the debugger from another tool chain to debug the code. Alternatively, the mechanism for interaction with the debugger can be replaced with something that uses one of the UARTs, eliminating (for the most part) the need for a debugger (other than for debugging).

Each of these aspects are discussed individually.

30.2 Compilers

There are four things that need special treatment between the various tool chains:

- How the compiler is invoked
- Compiler specific constructs
- Assembler specific constructs
- How code is linked

This discussion only applies to building from the command line; building with a project file utilizes the normal mechanisms for the GUI in question.

30.2.1 Invoking The Compiler

The `makedefs` file contains a set of rules for compiling C source files, compiling assembly source files, creating object libraries, and linking applications. These rules utilize the traditional variables for invoking the tools, such as `CC`, `CFLAGS`, and so on. These variables are given default values based on the tool chain being used; it is recommended that the variables that contain executable names be left alone and those that contain flags (such as `CFLAGS`) only be augmented.

All the rules place the targets into a tool chain-specific directory. For example, building a C source file with RealView Microcontroller Development Kit places the object file in the `rvmdk` directory; the linked application and/or object library would also go into the same directory. By doing this, the objects from multiple tool chains can exist simultaneously in the source tree without becoming intermingled.

Automatically generated dependencies are utilized by the rules as well. Most modern compilers support a `-MD` or similar option that causes it to write out a dependency file when compiling. In this manner, the dependencies are automatically generated when the file is first compiled, and

are regenerated whenever the file is recompiled (which would result if any of the dependencies changed, which might result in new dependencies). The dependencies are therefore always up to date. Dependency files are placed into the tool chain directory like the object files, and have a `.d` file name extension.

The Makefile rules have a set of special variables that control the how the applications is built. These take into account the tools being used to build the application as well as the target applications name so that a Makefile can build more than one application and have the same makefile. The link rules also have a set of variables that allow the linker to be uniquely configured for each application. In all of them are the base name of the application; for example, if the target is `foobar.axf` then the special variables would be `..._foobar`. The variables are:

`PART`	This is the Stellaris microcontroller for which the application is being built.
`ROOT`	This specifies the relative location of the base directory of the Stellaris Peripheral Driver Library installation. This is used to inform the build process where the rest of the peripheral driver library build tools are located.
`VPATH`	This variable allows the build process a search path to find source files that do not exist in this directory.
`IPATH`	This variable allows the build process a search path to find header files that do not exist in this directory.
`ENTRY_target`	This is the entry point for the application. Typically this is `ResetISR`.
`ROBASE_target`	This is the address to use for the base address of the read only area of the application. If this is undefined, then the value defaults to 0x0000.0000. If it is specified, this is the location of the first byte of the application. This is useful for moving the starting address of an application to an address other than the beginning of flash or to move the address to SRAM for applications that need to be linked to run from SRAM. This value is only used by the Keil tools as other tool chains support linker scripts that can provide this functionality. When this value is specified in the Makefile, `SCATTERtools_target` should not be specified as this results in conflicting linker commands and causes the build to fail.
`LDFLAGStools_target`	This contains tool chain specific linker flags that are also specific to the application. The `tools` portion is replaced with the tool chain to which the flags apply; so, for example, to supply additional linker flags to the RealView linker, use `LDFLAGSrvmdk_target`.
`SCATTERtools_target`	This is the name of the tool chain specific linker script used to link the application. Typically this is `../../../${COMPILER}/standalone.ld`.
`CFLAGStools`	This specifies any tool chain specific compiler options that need to be specified to compile the project.

With these rules, makefiles become a simple list of the targets to be built (either applications, libraries, or both), the object files that comprise the target, and a set of target-specific variables in the case of applications.

For the peripheral driver library itself (contained in the `src` directory), some special flags are passed to the compiler to place each global symbol (be it a variable or a function) into its own separate section. This makes it is possible to minimize the impact of using a driver; for example, using the UART in an output only mode with only the UARTConfigSetExpClk() and UARTCharPut() APIs being used, all the APIs for reading data, getting the configuration, and so on, do not get linked into the application (as they would if all of the globals were built into a single section).

30.2.2 Understanding Linker Scripts

This section covers the default linker scripts that are provided as part of the peripheral driver library release. This will cover the basics of the various settings in each of the linker scripts for all of the tool chains supported by peripheral driver library in order to help better understand how to use the linker scripts that are provided. It should be noted that the evaluation version of the Keil tools do not allow the use of linker scripts. Because of this none of the Keil builds use a linker script. Instead the build process generates the appropriate linker command line options to modify the address map of the application.

30.2.2.1 CodeSourcery GCC

The default linker script for this tool chain is located in the file `gcc/standalone.ld`. This file is broken down into two sections, the first section describes the memory available on the device and the second describes where to place the code and data for the application.

Note:
When using the CodeSourcery Sourcery G++ tool chain you also have the option of using CodeSourcery's method for installing interrupt handlers and specifying linker scripts. The "Getting Started" documentation provided with the CodeSourcery release describes how to use their tools to install interrupt handlers as well as how to use the linker scripts that are provide with their tools.

The rest of this section will cover the linker scripts provided by the peripheral driver library release.

```
MEMORY
{
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x00010000
    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00002000
}

SECTIONS
{
    .text :
    {
        _text = .;
        KEEP(*(.isr_vector))
        *(.text*)
        *(.rodata*)
        _etext = .;
```

```
    } > FLASH
    .data : AT (ADDR(.text) + SIZEOF(.text))
    {
        _data = .;
        *(vtable)
        *(.data*)
        _edata = .;
    } > SRAM
    .bss :
    {
        _bss = .;
        *(.bss*)
        *(COMMON)
        _ebss = .;
    } > SRAM}
```

The `MEMORY` section describes the amount of flash and SRAM available to the project. Each lines has an `ORIGIN` and a `LENGTH` value that set the amount of flash or SRAM available. In this case the flash is set to start at address 0x0000.0000 and have 64K bytes available. The SRAM is set to start at address 0x2000.0000 and have 8K bytes available.

The next part of the file, labeled `SECTION`, describes in detail where the code and data will be placed for the application. The default linker script has sections that are put in very specific places in order for the application to function correctly.

KEEP(*(.isr_vector)) - This statement places the read only interrupt vectors at the beginning of this section, which in this case is the beginning of flash due to the `FLASH` at the end of this section definition. This section should be at the beginning of flash in order for the application to boot correctly from the flash. The beginning of the default startup file for gcc, provided by the peripheral driver library, has the following code snippet to place the fixed interrupt handlers in the appropriate section.

```
__attribute__ ((section(".isr_vector")))
void (* const g_pfnVectors[])(void) =
{
    ...
}
```

***(.text*)** - Since .text is the default label applied to all “C” code, this statement places this read only code into the section following the interrupt vectors.

***(.rodata*)** - This section holds any constant read only data or the values for any initialized variables in the code. This section will normally immediately follow the .text read only code. This is important as any initialized values must be copied from flash to SRAM by the startup code.

_text = .; _etext = .; - These labels are inserted to allow application code to determine the size and the location of the read only area. These are accessible as global the following global variables:

```
extern unsigned long _text;
extern unsigned long _etext;
```

***(vtable)** - If an application uses the IntRegister() or IntUnregister() APIs, this entry places the vector table at the beginning of SRAM so that it can be modified by the APIs. This `vtable` label is attached to the code, or in this case data, by the following sequence in the file `DriverLib/src/interrupt.c`:

```
static __attribute__((section("vtable")))
void (*g_pfnRAMVectors[NUM_INTERRUPTS])(void);
```

∗**(.data**∗**)** - This section places all of the initialized read/write data after the SRAM based vector table. The `AT (ADDR(.text) + SIZEOF(.text))` actually modifies the load address to be at the end of the code section. This is where the actual initialization values for the variables are located. The actual run time address of the variables is located in SRAM. This allows the start up code to copy the initial data values from flash into the proper place in SRAM before executing the main application.

_data = .; _edata = .; - These labels are inserted to allow application code to determine the size and location of the initialized read/write data area. These values are accessible via the following global variables:

```
extern unsigned long _data;
extern unsigned long _edata;
```

∗**(.bss**∗**)** - This section contain all of the uninitialized data for the project. This usually includes the stack and other variables that are not set to any value by default.

_bss = .; _ebss = .; - These labels are inserted to allow application code to determine the size and location of the uninitialized read/write data area. These values are accessible via the following global variables:

```
extern unsigned long _bss;
extern unsigned long _ebss;
```

∗**(COMMON)** - Under some circumstances, gcc will place some global variables in this section. This requires that this section should be included to insure that these variables are correctly located in SRAM.

30.2.2.2 Keil RV-MDK

The default linker script file is located in the `rvmdk/standalone.sct`. This file cannot be used with the evaluation version of the tool chain. Like the previous file format, this file is broken down into each value in the following example:

```
LR_IROM 0x00000000 0x00010000
{
    ;
    ; Specify the Execution Address of the code and the size.
    ;
    ER_IROM 0x00000000 0x00010000
    {
        *.o (RESET, +First)
        * (+RO)
    }

    ;
    ; Specify the Execution Address of the data area.
    ;
    RW_IRAM 0x20000000 0x00002000
    {
        * (+RW)
        * (+ZI)
    }
}
```

∗**.o (RESET, +First)** - This use of `RESET` allows the fixed vectors to be located at the beginning of flash. The code example below is taken from the default `rvmdk/Startup.s` that is used by the example code provided with the peripheral driver library.

```
        AREA    RESET, CODE, READONLY
        THUMB
Vectors
        DCD     StackMem + Stack            ; Top of Stack
        DCD     Reset_Handler               ; Reset Handler
...
```

* **(+RO)** - This section places all of the code and read only data at the beginning of flash. This section will also hold any constant read only data or the initial values for any variables in the code that have an initial value and it will immediately follow the `RESET` code section. This section must be located in flash since any initialized values must be copied from flash to SRAM by the startup code.

* **(+RW)** - This section places all the initialized read/write data after the modifiable vector table. The Keil "C" start up code takes care of copying the constant initializers from flash to this location in SRAM.

* **(+ZI)** - This section contain all of the uninitialized data for the project. This usually includes the stack and other variables that are not set to any value by default.

30.2.2.3 IAR EW-ARM

This default linker script file is located in the file `ewarm/standalone.xcl`. Unlike the other tool chains this linker script is written so that it is passed as command line options to the linker and not in a more formal linker script format. However each section label that is used by the peripheral driver library and the example code is covered in this section.

```
//
// Set the CPU type to ARM.
//
-carm

//
// Define the size of flash and SRAM.
//
-DROMSTART=00000000
-DROMEND=0000FFFF
-DRAMSTART=20000000
-DRAMEND=20001FFF

//
// Define the sections to place into flash, and the order to place them.
//
-Z(CODE)INTVEC=ROMSTART-ROMEND
-Z(CODE)ICODE,DIFUNCT=ROMSTART-ROMEND
-Z(CODE)CODE=ROMSTART-ROMEND
-Z(CONST)CODE_ID=ROMSTART-ROMEND
-Z(CONST)INITTAB,DATA_ID,DATA_C=ROMSTART-ROMEND
-Z(CONST)CHECKSUM=ROMSTART-ROMEND
```

INTVEC - This section holds the vector table for the application and should be located at the beginning of FLASH. The code example below shows how the default start up file, provided by the peripheral driver library, marks the vector table as belonging to this section.

```
__root const uVectorEntry g_pfnVectors[] @ "INTVEC" =
{
    { .ulPtr = (unsigned long)pulStack + sizeof(pulStack) },
                                            // The initial stack pointer
```

```
    ResetISR,                               // The reset handler
    ...
```

ICODE - This section holds the start up code or any of the handlers for the exceptions.

CODE - This section holds the remainder of the application code that will be used, minus any code that is has been marked to be run directly from SRAM.

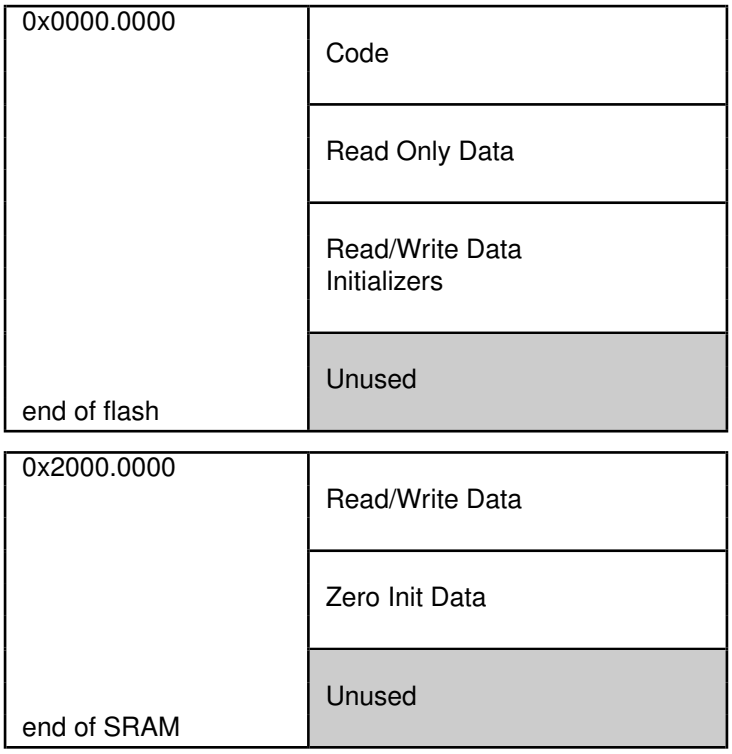
CODE_ID - This section holds any code that was designated to be run directly from SRAM. These sections are marked by using the `__ramfunc` in "C" code. This type of section is not used by the peripheral driver library or any examples provided with the peripheral driver library.

INITTAB,DATA_ID,DATA_C - These sections holds constant or global data that has been copied from flash.

CHECKSUM - This section is not used by the peripheral driver library or any of the provided examples.

30.2.2.4 Default Memory Map

The following is the default memory map of a peripheral driver library application, regardless of the tool chain being used.



30.2.3 Compiler Constructs

Occasionally there is a need to use compiler-specific constructs in C source files. When that need arises, two options are available:

- Provide separate versions of the source file for each tool chain. This has been done with the boot code; it is basically identical from tool chain to tool chain, the exception being the construct used to tag the vector table for placement at the beginning of flash and the names of the symbols created by the linker as it creates the “code”, “data”, and “bss” segments.
- Use `#ifdef`/`#endif` around the constructs specific to each tool chain.

When providing separate files, the pathname of the file should contain the value of `${COMPILER}` somewhere in it; either as a directory name or as part of the file name. This way, a dependency in the Makefile can utilize the value of the `${COMPILER}` to cause the correct version of the file to be used. In the examples provided, this can be seen in the boot code; separate versions are provided for each tool chain supported. The correct version of the boot code is found via the `${COMPILER}` in the Makefile for the boot code file name.

When using `#ifdef`/`#endif`, the value of `${COMPILER}` again comes into play. Each source file is built with a `-D${COMPILER}` passed to the compiler, so the value of the `${COMPILER}` variable can be used in a `#ifdef` to include compiler specific code. This is not the preferred method since it is very error prone; if it is used to include a small piece of code within a function (for example), it would be too easy to forget about that when porting to a different tool chain which would result in that small piece of code not appearing in the object produced by new tool chain. In the first method, the file simply would not exist and a build error would occur.

30.2.4 Assembler Constructs

The macros in `asmdefs.h` hide the differences in syntax and directives between the assemblers of different tool chains. By utilizing these macros, the assembly files are free of `#ifdef toolchain` constructs, making them easier to understand and maintain. The following macros are provided for use in writing assembler independent source files:

`__ALIGN__`	This is used to place the next item on a four-byte alignment boundary in memory.
`__BSS__`	This is used to indicate that the items that follow should be placed into the “bss” segment of the executable. These items have storage space reserved but will not have initializers supplied in the executable, instead depending upon the boot code to zero fill the storage.
`__DATA__`	This is used to indicate that the items that follow should be placed into the “data” segment of the executable. These items have storage space reserved in SRAM, the initializers placed in flash, and the initializers copied from SRAM to flash by the boot code.
`__END__`	This indicates that the end of the assembly source file has been reached.
`__EXPORT__`	This indicates that a label should be made available to object files outside the current source file.
`__IMPORT__`	This indicates that a label from another object file is to be referenced from this source file.

__LABEL__	This provides a symbolic name for the current location. A label can be used as a branch target or to load/store data. Note that labels are not accessible outside the current source file unless exported with __EXPORT__.
__STR__	This is used to declare a string of data (that is, a zero terminated sequence of bytes).
__TEXT__	This is used to indicate that the items that follow should be placed into the "text" segment of the executable. This must be used before all code so that it are located correctly.
__THUMB_LABEL__	This indicates that the next label (which must immediately follow) is a Thumb label. All labels must be marked as Thumb labels or they will not work properly as a branch target.
__WORD__	This is used to declare a word (32-bits) of data.

asmdefs.h must included before anything else in the assembly language source file since it includes some common setup pseudo-ops that are required to put the assembler into the correct mode; failure to do so may cause the assembler to behave incorrectly.

30.2.5 Linking Applications

When linking applications, each global entity needs to be placed into the appropriate portion of memory in order for the application to work correctly. Some things must be placed at specific locations (such as the default vector table, which must reside at 0x0000.0000). Other things must be placed into the correct portion of memory (all code needs to be placed in flash and all read/write data in SRAM).

A linker script is used to perform this task. Linker scripts are not portable between tool chains, so separate versions are supplied for each tool chain; they are in the <toolchain>/standalone.ld files (standalone.xcl in the case of IAR Embedded Workbench). These linker scripts are fairly simple; they place all the code in flash (the "code" segment), all the read/writes in SRAM (the "data" and "bss" segments), the "data" segment initializers in flash at the end of the "code" segment, the read-only vector table at the beginning of flash, and the read/write vector table from the interrupt driver (if it is used) at the beginning of SRAM. The boot code in <toolchain>/startup.c depends upon this layout of the memory; if the memory layout is changed then both file may need changes (or replacement).

30.3 Debuggers

Debuggers typically have a method for code running on the target to interact with the debugger: to read/write host files, print messages on the debugger console, and so on. These mechanisms have been abstracted into a set of functions that can be called by an application without regard to the debugger that they are using. These functions are discussed in chapter 27; they are the Diag... functions.

The debugger interface code resides in a file called utils/${DEBUGGER}.S (or .c if implemented

in C). The rules in the makefile specify a dependency on `${DEBUGGER}.o`; therefore, by changing the value of `${DEBUGGER}`, the debugger interface code is changed. This is what allows the compiler from one tool chain and the debugger from another to be used together (assuming of course that they support the same executable file format); `${COMPILER}` specifies the tools used to build the code and `${DEBUGGER}` specifies the debugger interface to use.

Several interesting things can be done with this interface:

- A serial version could be created, where files are not supported but stdio is supported. All stdio operations would go to a UART.
- A serial memory version could be created. The application could then be developed using host files via the debugger (where the file contents are much easier to examine) and then switched to use a serial memory version when appropriate.
- A stub version could be created where each function is a NOP. This would eliminate all debugger interaction from the application.
- A debug version could be created, where it normally acts as a NOP but if turned on via a special flag would start outputting stdio to a defined place (such as an unused UART). This would allow tracing capabilities to be left in production code; it would normally do nothing (giving customers no clues as to what it is doing/how it is doing things) but could be enabled by field support personnel to help determine why failures are occurring.

31 DK-LM3S101 Example Applications

31.1 Introduction

The DK-LM3S101 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the Peripheral Device Controller on the Stellaris Family Development Kit board. The PDC is used to access the character LCD, eight user LEDs, eight user DIP switches, and twenty-four GPIOs.

There is an IAR workspace file (`dk-lm3s101.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`dk-lm3s101-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`dk-lm3s101.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/dk-lm3s101` subdirectory of the peripheral driver library source distribution.

31.2 API Functions

Functions

- unsigned char PDCDIPRead (void)
- unsigned char PDCGPIODirRead (unsigned char ucIdx)
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)
- unsigned char PDCGPIORead (unsigned char ucIdx)
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)
- void PDCInit (void)
- void PDCLCDBacklightOff (void)
- void PDCLCDBacklightOn (void)
- void PDCLCDClear (void)
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)
- void PDCLCDInit (void)
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)

- unsigned char PDCLEDRead (void)
- void PDCLEDWrite (unsigned char ucLED)
- unsigned char PDCRead (unsigned char ucAddr)
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)

31.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

31.2.2 Function Documentation

31.2.2.1 PDCDIPRead

Read the current value of the PDC DIP switches.

Prototype:

```
unsigned char
PDCDIPRead(void)
```

Description:

This function will read the current value of the DIP switches attached to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The current state of the DIP switches.

31.2.2.2 PDCGPIODirRead

Reads a GPIO direction register.

Prototype:

```
unsigned char
PDCGPIODirRead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO direction registers in the PDC. The direction bit is set for pins that are outputs and clear for pins that are inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the direction register.

31.2.2.3 PDCGPIODirWrite

Write a GPIO direction register.

Prototype:

```
void
PDCGPIODirWrite(unsigned char ucIdx,
                unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO direction register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO direction register.

Description:

This function writes ones of the GPIO direction registers in the PDC. The direction bit should be set for pins that are to be outputs and clear for pins that are to be inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

31.2.2.4 PDCGPIORead

Reads a GPIO data register.

Prototype:

```
unsigned char
PDCGPIORead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO data registers in the PDC. The value returned for a pin is the value being driven out for outputs or the value being read for inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the data register.

31.2.2.5 PDCGPIOWrite

Write a GPIO data register.

Prototype:

```
void
PDCGPIOWrite(unsigned char ucIdx,
             unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO data register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO data register.

Description:

This function writes one of the GPIO direction registers in the PDC. The written to a pin is driven out for output pins and ignored for input pins.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

31.2.2.6 PDCInit

Initializes the connection to the PDC.

Prototype:

```
void
PDCInit(void)
```

Description:

This function will enable clocking to the SSI and GPIO A modules, configure the GPIO pins to be used for an SSI interface, and it will configure the SSI as a 1 Mbps master device, operating in MOTO mode. It will also enable the SSI module, and will enable the chip select for the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

31.2.2.7 PDCLCDBacklightOff

Turn off the backlight.

Prototype:

```
void
PDCLCDBacklightOff(void)
```

Description:

This function turns off the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

31.2.2.8 PDCLCDBacklightOn

Turns on the backlight.

Prototype:

```
void
PDCLCDBacklightOn(void)
```

Description:

This function turns on the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

31.2.2.9 PDCLCDClear

Clear the screen.

Prototype:

```
void
PDCLCDClear(void)
```

Description:

This function clears the contents of the LCD screen. The cursor will be returned to the upper left corner.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

31.2.2.10 PDCLCDCreateChar

Write a character pattern to the LCD.

Prototype:

```
void
PDCLCDCreateChar(unsigned char ucChar,
                 unsigned char *pucData)
```

Parameters:

ucChar is the character index to create. Valid values are zero through seven.

pucData is the data for the character pattern. It contains eight bytes, with the first byte being the top row of the pattern. In each byte, the LSB is the right pixel of the pattern.

Description:

This function will write a character pattern into the LCD for use as a character to be displayed. After writing the pattern, it can be used on the LCD by writing the corresponding character index to the display.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

31.2.2.11 PDCLCDInit

Initializes the LCD display.

Prototype:
```
void
PDCLCDInit(void)
```

Description:
This function will set up the LCD display for writing. It will set the data bus to 8 bits, set the number of lines to 2, and the font size to 5x10. It will also turn the display off, clear the display, turn the display back on, and enable the backlight.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Note:
The PDC must be initialized via the PDCInit() function before this function can be called. Also, it may be necessary to adjust the contrast potentiometer in order to discern any output on the LCD display.

Returns:
None.

31.2.2.12 PDCLCDSetPos

Set the position of the cursor.

Prototype:
```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

Parameters:
ucX is the horizontal position. Valid values are zero through fifteen.
ucY is the vertical position.. Valid values are zero and one.

Description:
This function will move the cursor to the specified position. All characters written to the LCD are placed at the current cursor position, which is automatically advanced.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

31.2.2.13 PDCLCDWrite

Writes a string to the LCD display.

Prototype:

```
void
PDCLCDWrite(const char *pcStr,
            unsigned long ulCount)
```

Parameters:

pcStr pointer to the string to be displayed.

ulCount is the number of characters to be displayed.

Description:

This function will display a string on the LCD at the current cursor position. It is the caller's responsibility to position the cursor to the place where the string should be displayed (either explicitly via PDCLCDSetPos() or implicitly from where the cursor was left after a previous call to PDCLCDWrite()), and to properly account for the LCD boundary (line wrapping is not automatically performed). Null characters are not treated special and are written to the LCD, which interprets it as a special programmable character glyph (see PDCLCDCreateChar()).

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

31.2.2.14 PDCLEDRead

Read the current status of the PDC LEDs.

Prototype:

```
unsigned char
PDCLEDRead(void)
```

Description:

This function will read the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The value currently displayed by the LEDs.

31.2.2.15 PDCLEDWrite

Write to the PDC LEDs.

Prototype:

```
void
PDCLEDWrite(unsigned char ucLED)
```

Parameters:

ucLED value to write to the LEDs.

Description:

This function set the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

31.2.2.16 PDCRead

Read a PDC register.

Prototype:

```
unsigned char
PDCRead(unsigned char ucAddr)
```

Parameters:

ucAddr specifies the PDC register to read.

Description:

This function will perform the SSI transfers required to read a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

Returns the value read from the PDC.

31.2.2.17 PDCWrite

Write a PDC register.

Prototype:

```
void
PDCWrite(unsigned char ucAddr,
         unsigned char ucData)
```

Parameters:

ucAddr specifies the PDC register to write.

ucData specifies the data to write.

Description:

This function will perform the SSI transfers required to write a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

31.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

Comparator (comparator)

This example application demonstrates the operation of the analog comparator(s). Comparator zero (which is present on all devices that have analog comparators) is configured to compare its negative input to an internally generated 1.65 V reference and toggle the state of the LED on port B0 based on comparator change interrupts. The LED will be turned on by the interrupt handler when a rising edge on the comparator output is detected, and will be turned off when a falling edge is detected.

In order for this example to work properly, the ULED0 (JP22) jumper must be installed on the board.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 of Sandstorm-class Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata is fixed in Rev C2 of Sandstorm-class Stellaris microcontrollers.

GPIO (gpio_led)

This example application uses LEDs connected to GPIO pins to create a “roving eye” display. Port B0-B3 are driven in a sequential manner to give the illusion of an eye looking back and forth.

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), ULED2 (JP24), and ULED3 (JP25) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the LCD and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; individual LEDs connected to port B0-B2 will be turned on upon interrupt handler entry and off before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), and ULED2 (JP24) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

DK-LM3S101 Quickstart Application (qs_dk-lm3s101)

This example uses the photocell on the development board to create a geiger counter for visible light. In bright light, the click rate (that is, the count) increases; in low light it decreases. The light reading is also displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn off the clicking noise on and off; when off the LCD and UART still provide the light reading.

In the default jumper configuration of the development board, this example actually samples the potentiometer and the push button will not work. In order for this example to fully work, the following jumper wire connections must be made: JP3 pin 1 to JP5 pin 2 (requiring the removal of the jumper on JP5) and JP19 pin 2 to J6 pin 6.

SSI (ssi_atmel)

This example application uses the SSI master to communicate with the Atmel AT25F1024A EEPROM that is on the development board. The first 256 bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the SSI interrupt; since a 256-byte read at a 1 MHz SSI bus speed takes around 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own GPIO (port B0 and B1) on each interrupt; the attached LED will indicate the occurrence and rate of interrupts.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (the SER0 connector on the Stellaris Family Development Board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port B0 is inverted so that it is easy to see that it is being fed, which occurs once every second.

32 DK-LM3S102 Example Applications

32.1 Introduction

The DK-LM3S102 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the Peripheral Device Controller on the Stellaris Family Development Kit board. The PDC is used to access the character LCD, eight user LEDs, eight user DIP switches, and twenty-four GPIOs.

There is an IAR workspace file (`dk-lm3s102.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`dk-lm3s102-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`dk-lm3s102.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/dk-lm3s102` subdirectory of the peripheral driver library source distribution.

32.2 API Functions

Functions

- unsigned char PDCDIPRead (void)
- unsigned char PDCGPIODirRead (unsigned char ucIdx)
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)
- unsigned char PDCGPIORead (unsigned char ucIdx)
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)
- void PDCInit (void)
- void PDCLCDBacklightOff (void)
- void PDCLCDBacklightOn (void)
- void PDCLCDClear (void)
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)
- void PDCLCDInit (void)
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)

- unsigned char PDCLEDRead (void)
- void PDCLEDWrite (unsigned char ucLED)
- unsigned char PDCRead (unsigned char ucAddr)
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)

32.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

32.2.2 Function Documentation

32.2.2.1 PDCDIPRead

Read the current value of the PDC DIP switches.

Prototype:

```
unsigned char
PDCDIPRead(void)
```

Description:

This function will read the current value of the DIP switches attached to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The current state of the DIP switches.

32.2.2.2 PDCGPIODirRead

Reads a GPIO direction register.

Prototype:

```
unsigned char
PDCGPIODirRead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO direction registers in the PDC. The direction bit is set for pins that are outputs and clear for pins that are inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the direction register.

32.2.2.3 PDCGPIODirWrite

Write a GPIO direction register.

Prototype:

```
void
PDCGPIODirWrite(unsigned char ucIdx,
                unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO direction register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO direction register.

Description:

This function writes ones of the GPIO direction registers in the PDC. The direction bit should be set for pins that are to be outputs and clear for pins that are to be inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

32.2.2.4 PDCGPIORead

Reads a GPIO data register.

Prototype:

```
unsigned char
PDCGPIORead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO data registers in the PDC. The value returned for a pin is the value being driven out for outputs or the value being read for inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the data register.

32.2.2.5 PDCGPIOWrite

Write a GPIO data register.

Prototype:

```
void
PDCGPIOWrite(unsigned char ucIdx,
             unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO data register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO data register.

Description:

This function writes one of the GPIO direction registers in the PDC. The written to a pin is driven out for output pins and ignored for input pins.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

32.2.2.6 PDCInit

Initializes the connection to the PDC.

Prototype:

```
void
PDCInit(void)
```

Description:

This function will enable clocking to the SSI and GPIO A modules, configure the GPIO pins to be used for an SSI interface, and it will configure the SSI as a 1 Mbps master device, operating in MOTO mode. It will also enable the SSI module, and will enable the chip select for the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

32.2.2.7 PDCLCDBacklightOff

Turn off the backlight.

Prototype:

```
void
PDCLCDBacklightOff(void)
```

Description:

This function turns off the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

32.2.2.8 PDCLCDBacklightOn

Turns on the backlight.

Prototype:

```
void
PDCLCDBacklightOn(void)
```

Description:

This function turns on the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

32.2.2.9 PDCLCDClear

Clear the screen.

Prototype:

```
void
PDCLCDClear(void)
```

Description:

This function clears the contents of the LCD screen. The cursor will be returned to the upper left corner.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

32.2.2.10 PDCLCDCreateChar

Write a character pattern to the LCD.

Prototype:

```
void
PDCLCDCreateChar(unsigned char ucChar,
                 unsigned char *pucData)
```

Parameters:

ucChar is the character index to create. Valid values are zero through seven.

pucData is the data for the character pattern. It contains eight bytes, with the first byte being the top row of the pattern. In each byte, the LSB is the right pixel of the pattern.

Description:

This function will write a character pattern into the LCD for use as a character to be displayed. After writing the pattern, it can be used on the LCD by writing the corresponding character index to the display.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

32.2.2.11 PDCLCDInit

Initializes the LCD display.

Prototype:
```
void
PDCLCDInit(void)
```

Description:
This function will set up the LCD display for writing. It will set the data bus to 8 bits, set the number of lines to 2, and the font size to 5x10. It will also turn the display off, clear the display, turn the display back on, and enable the backlight.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Note:
The PDC must be initialized via the PDCInit() function before this function can be called. Also, it may be necessary to adjust the contrast potentiometer in order to discern any output on the LCD display.

Returns:
None.

32.2.2.12 PDCLCDSetPos

Set the position of the cursor.

Prototype:
```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

Parameters:
ucX is the horizontal position. Valid values are zero through fifteen.
ucY is the vertical position.. Valid values are zero and one.

Description:
This function will move the cursor to the specified position. All characters written to the LCD are placed at the current cursor position, which is automatically advanced.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

32.2.2.13 PDCLCDWrite

Writes a string to the LCD display.

Prototype:

```
void
PDCLCDWrite(const char *pcStr,
            unsigned long ulCount)
```

Parameters:

pcStr pointer to the string to be displayed.

ulCount is the number of characters to be displayed.

Description:

This function will display a string on the LCD at the current cursor position. It is the caller's responsibility to position the cursor to the place where the string should be displayed (either explicitly via PDCLCDSetPos() or implicitly from where the cursor was left after a previous call to PDCLCDWrite()), and to properly account for the LCD boundary (line wrapping is not automatically performed). Null characters are not treated special and are written to the LCD, which interprets it as a special programmable character glyph (see PDCLCDCreateChar()).

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

32.2.2.14 PDCLEDRead

Read the current status of the PDC LEDs.

Prototype:

```
unsigned char
PDCLEDRead(void)
```

Description:

This function will read the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The value currently displayed by the LEDs.

32.2.2.15 PDCLEDWrite

Write to the PDC LEDs.

Prototype:

```
void
PDCLEDWrite(unsigned char ucLED)
```

Parameters:

ucLED value to write to the LEDs.

Description:

This function set the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

32.2.2.16 PDCRead

Read a PDC register.

Prototype:

```
unsigned char
PDCRead(unsigned char ucAddr)
```

Parameters:

ucAddr specifies the PDC register to read.

Description:

This function will perform the SSI transfers required to read a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

Returns the value read from the PDC.

32.2.2.17 PDCWrite

Write a PDC register.

Prototype:

```
void
PDCWrite(unsigned char ucAddr,
         unsigned char ucData)
```

Parameters:

ucAddr specifies the PDC register to write.

ucData specifies the data to write.

Description:

This function will perform the SSI transfers required to write a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

32.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

Comparator (comparator)

This example application demonstrates the operation of the analog comparator(s). Comparator zero (which is present on all devices that have analog comparators) is configured to compare its negative input to an internally generated 1.65 V reference and toggle the state of the LED on port B0 based on comparator change interrupts. The LED will be turned on by the interrupt handler when a rising edge on the comparator output is detected, and will be turned off when a falling edge is detected.

In order for this example to work properly, the ULED0 (JP22) jumper must be installed on the board.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 of Sandstorm-class Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata is fixed in Rev C2 of Sandstorm-class Stellaris microcontrollers.

GPIO (gpio_led)

This example application uses LEDs connected to GPIO pins to create a “roving eye” display. Port B0-B3 are driven in a sequential manner to give the illusion of an eye looking back and forth.

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), ULED2 (JP24), and ULED3 (JP25) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the LCD and is a starting point for more complicated applications.

I2C (i2c_atmel)

This example application uses the I2C master to communicate with the Atmel AT24C08A EEPROM that is on the development board. The first sixteen bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the I2C interrupt; since a sixteen-byte read at a 100 kHz I2C bus speed takes almost 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

In order for this example to work properly, the I2C_SCL (JP14), I2C_SDA (JP13), and I2CM_A2 (JP11) jumpers must be installed on the board, and the I2CM_WP (JP12) jumper must be removed.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; individual LEDs connected to port B0-B2 will be turned on upon interrupt handler entry and off before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), and ULED2 (JP24) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

DK-LM3S102 Quickstart Application (qs_dk-lm3s102)

This example uses the photocell on the development board to create a geiger counter for visible light. In bright light, the click rate (that is, the count) increases; in low light it decreases. The light reading is also displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn off the clicking noise on and off; when off the LCD and UART still provide the light reading.

In the default jumper configuration of the development board, this example actually samples the potentiometer and the push button will not work. In order for this example to fully work, the following jumper wire connections must be made: JP3 pin 1 to JP5 pin 2 (requiring the removal of the jumper on JP5) and JP19 pin 2 to J6 pin 6.

SSI (ssi_atmel)

This example application uses the SSI master to communicate with the Atmel AT25F1024A EEPROM that is on the development board. The first 256 bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the SSI interrupt; since a 256-byte read at a 1 MHz SSI bus speed takes around 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own GPIO (port B0 and B1) on each interrupt; the attached LED will indicate the occurrence and rate of interrupts.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (the SER0 connector on the Stellaris Family Development Board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port B0 is inverted so that it is easy to see that it is being fed, which occurs once every second.

33 DK-LM3S301 Example Applications

33.1 Introduction

The DK-LM3S301 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the Peripheral Device Controller on the Stellaris Family Development Kit board. The PDC is used to access the character LCD, eight user LEDs, eight user DIP switches, and twenty-four GPIOs.

There is an IAR workspace file (`dk-lm3s301.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`dk-lm3s301-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`dk-lm3s301.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/dk-lm3s301` subdirectory of the peripheral driver library source distribution.

33.2 API Functions

Functions

- unsigned char PDCDIPRead (void)
- unsigned char PDCGPIODirRead (unsigned char ucIdx)
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)
- unsigned char PDCGPIORead (unsigned char ucIdx)
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)
- void PDCInit (void)
- void PDCLCDBacklightOff (void)
- void PDCLCDBacklightOn (void)
- void PDCLCDClear (void)
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)
- void PDCLCDInit (void)
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)

- unsigned char PDCLEDRead (void)
- void PDCLEDWrite (unsigned char ucLED)
- unsigned char PDCRead (unsigned char ucAddr)
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)

33.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

33.2.2 Function Documentation

33.2.2.1 PDCDIPRead

Read the current value of the PDC DIP switches.

Prototype:

```
unsigned char
PDCDIPRead(void)
```

Description:

This function will read the current value of the DIP switches attached to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The current state of the DIP switches.

33.2.2.2 PDCGPIODirRead

Reads a GPIO direction register.

Prototype:

```
unsigned char
PDCGPIODirRead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO direction registers in the PDC. The direction bit is set for pins that are outputs and clear for pins that are inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the direction register.

33.2.2.3 PDCGPIODirWrite

Write a GPIO direction register.

Prototype:

```
void
PDCGPIODirWrite(unsigned char ucIdx,
                unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO direction register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO direction register.

Description:

This function writes ones of the GPIO direction registers in the PDC. The direction bit should be set for pins that are to be outputs and clear for pins that are to be inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

33.2.2.4 PDCGPIORead

Reads a GPIO data register.

Prototype:

```
unsigned char
PDCGPIORead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO data registers in the PDC. The value returned for a pin is the value being driven out for outputs or the value being read for inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the data register.

33.2.2.5 PDCGPIOWrite

Write a GPIO data register.

Prototype:

```
void
PDCGPIOWrite(unsigned char ucIdx,
             unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO data register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO data register.

Description:

This function writes one of the GPIO direction registers in the PDC. The written to a pin is driven out for output pins and ignored for input pins.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

33.2.2.6 PDCInit

Initializes the connection to the PDC.

Prototype:

```
void
PDCInit(void)
```

Description:

This function will enable clocking to the SSI and GPIO A modules, configure the GPIO pins to be used for an SSI interface, and it will configure the SSI as a 1 Mbps master device, operating in MOTO mode. It will also enable the SSI module, and will enable the chip select for the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

33.2.2.7 PDCLCDBacklightOff

Turn off the backlight.

Prototype:

```
void
PDCLCDBacklightOff(void)
```

Description:

This function turns off the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

33.2.2.8 PDCLCDBacklightOn

Turns on the backlight.

Prototype:

```
void
PDCLCDBacklightOn(void)
```

Description:

This function turns on the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

33.2.2.9 PDCLCDClear

Clear the screen.

Prototype:

```
void
PDCLCDClear(void)
```

Description:

This function clears the contents of the LCD screen. The cursor will be returned to the upper left corner.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

33.2.2.10 PDCLCDCreateChar

Write a character pattern to the LCD.

Prototype:

```
void
PDCLCDCreateChar(unsigned char ucChar,
                 unsigned char *pucData)
```

Parameters:

ucChar is the character index to create. Valid values are zero through seven.

pucData is the data for the character pattern. It contains eight bytes, with the first byte being the top row of the pattern. In each byte, the LSB is the right pixel of the pattern.

Description:

This function will write a character pattern into the LCD for use as a character to be displayed. After writing the pattern, it can be used on the LCD by writing the corresponding character index to the display.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

33.2.2.11 PDCLCDInit

Initializes the LCD display.

Prototype:
```
void
PDCLCDInit(void)
```

Description:
This function will set up the LCD display for writing. It will set the data bus to 8 bits, set the number of lines to 2, and the font size to 5x10. It will also turn the display off, clear the display, turn the display back on, and enable the backlight.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Note:
The PDC must be initialized via the PDCInit() function before this function can be called. Also, it may be necessary to adjust the contrast potentiometer in order to discern any output on the LCD display.

Returns:
None.

33.2.2.12 PDCLCDSetPos

Set the position of the cursor.

Prototype:
```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

Parameters:
ucX is the horizontal position. Valid values are zero through fifteen.
ucY is the vertical position.. Valid values are zero and one.

Description:
This function will move the cursor to the specified position. All characters written to the LCD are placed at the current cursor position, which is automatically advanced.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

33.2.2.13 PDCLCDWrite

Writes a string to the LCD display.

Prototype:

```
void
PDCLCDWrite(const char *pcStr,
            unsigned long ulCount)
```

Parameters:

pcStr pointer to the string to be displayed.

ulCount is the number of characters to be displayed.

Description:

This function will display a string on the LCD at the current cursor position. It is the caller's responsibility to position the cursor to the place where the string should be displayed (either explicitly via PDCLCDSetPos() or implicitly from where the cursor was left after a previous call to PDCLCDWrite()), and to properly account for the LCD boundary (line wrapping is not automatically performed). Null characters are not treated special and are written to the LCD, which interprets it as a special programmable character glyph (see PDCLCDCreateChar()).

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

33.2.2.14 PDCLEDRead

Read the current status of the PDC LEDs.

Prototype:

```
unsigned char
PDCLEDRead(void)
```

Description:

This function will read the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The value currently displayed by the LEDs.

33.2.2.15 PDCLEDWrite

Write to the PDC LEDs.

Prototype:

```
void
PDCLEDWrite(unsigned char ucLED)
```

Parameters:

ucLED value to write to the LEDs.

Description:

This function set the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

33.2.2.16 PDCRead

Read a PDC register.

Prototype:

```
unsigned char
PDCRead(unsigned char ucAddr)
```

Parameters:

ucAddr specifies the PDC register to read.

Description:

This function will perform the SSI transfers required to read a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

Returns the value read from the PDC.

33.2.2.17 PDCWrite

Write a PDC register.

Prototype:

```
void
PDCWrite(unsigned char ucAddr,
         unsigned char ucData)
```

Parameters:

ucAddr specifies the PDC register to write.

ucData specifies the data to write.

Description:

This function will perform the SSI transfers required to write a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

33.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

Comparator (comparator)

This example application demonstrates the operation of the analog comparator(s). Comparator zero (which is present on all devices that have analog comparators) is configured to compare its negative input to an internally generated 1.65 V reference and toggle the state of the LED on port B0 based on comparator change interrupts. The LED will be turned on by the interrupt handler when a rising edge on the comparator output is detected, and will be turned off when a falling edge is detected.

In order for this example to work properly, the ULED0 (JP22) jumper must be installed on the board.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 of Sandstorm-class Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata is fixed in Rev C2 of Sandstorm-class Stellaris microcontrollers.

GPIO (gpio_led)

This example application uses LEDs connected to GPIO pins to create a “roving eye” display. Port B0-B3 are driven in a sequential manner to give the illusion of an eye looking back and forth.

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), ULED2 (JP24), and ULED3 (JP25) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the LCD and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; individual LEDs connected to port B0-B2 will be turned on upon interrupt handler entry and off before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), and ULED2 (JP24) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 50 kHz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

DK-LM3S301 Quickstart Application (qs_dk-lm3s301)

This example uses the photocell on the development board to create a geiger counter for visible light. In bright light, the click rate (that is, the count) increases; in low light it decreases. The light reading is also displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn off the clicking noise on and off; when off the LCD and UART still provide the light reading.

In the default jumper configuration of the development board, this example actually samples the potentiometer and the push button will not work. In order for this example to fully work, the following jumper wire connections must be made: JP3 pin 1 to JP5 pin 2 (requiring the removal of the jumper on JP5) and JP19 pin 2 to J6 pin 6.

SSI (ssi_atmel)

This example application uses the SSI master to communicate with the Atmel AT25F1024A EEPROM that is on the development board. The first 256 bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the SSI interrupt; since a 256-byte read at a 1 MHz SSI bus speed takes around 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own GPIO (port B0 and B1) on each interrupt; the attached LED will indicate the occurrence and rate of interrupts.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (the SER0 connector on the Stellaris Family Development Board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port B0 is inverted so that it is easy to see that it is being fed, which occurs once every second.

34 DK-LM3S801 Example Applications

34.1 Introduction

The DK-LM3S801 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the Peripheral Device Controller on the Stellaris Family Development Kit board. The PDC is used to access the character LCD, eight user LEDs, eight user DIP switches, and twenty-four GPIOs.

There is an IAR workspace file (`dk-lm3s801.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`dk-lm3s801-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`dk-lm3s801.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/dk-lm3s801` subdirectory of the peripheral driver library source distribution.

34.2 API Functions

Functions

- unsigned char PDCDIPRead (void)
- unsigned char PDCGPIODirRead (unsigned char ucIdx)
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)
- unsigned char PDCGPIORead (unsigned char ucIdx)
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)
- void PDCInit (void)
- void PDCLCDBacklightOff (void)
- void PDCLCDBacklightOn (void)
- void PDCLCDClear (void)
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)
- void PDCLCDInit (void)
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)

- unsigned char PDCLEDRead (void)
- void PDCLEDWrite (unsigned char ucLED)
- unsigned char PDCRead (unsigned char ucAddr)
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)

34.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

34.2.2 Function Documentation

34.2.2.1 PDCDIPRead

Read the current value of the PDC DIP switches.

Prototype:

```
unsigned char
PDCDIPRead(void)
```

Description:

This function will read the current value of the DIP switches attached to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The current state of the DIP switches.

34.2.2.2 PDCGPIODirRead

Reads a GPIO direction register.

Prototype:

```
unsigned char
PDCGPIODirRead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO direction registers in the PDC. The direction bit is set for pins that are outputs and clear for pins that are inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the direction register.

34.2.2.3 PDCGPIODirWrite

Write a GPIO direction register.

Prototype:

```
void
PDCGPIODirWrite(unsigned char ucIdx,
                unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO direction register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO direction register.

Description:

This function writes ones of the GPIO direction registers in the PDC. The direction bit should be set for pins that are to be outputs and clear for pins that are to be inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

34.2.2.4 PDCGPIORead

Reads a GPIO data register.

Prototype:

```
unsigned char
PDCGPIORead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO data registers in the PDC. The value returned for a pin is the value being driven out for outputs or the value being read for inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the data register.

34.2.2.5 PDCGPIOWrite

Write a GPIO data register.

Prototype:

```
void
PDCGPIOWrite(unsigned char ucIdx,
             unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO data register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO data register.

Description:

This function writes one of the GPIO direction registers in the PDC. The written to a pin is driven out for output pins and ignored for input pins.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

34.2.2.6 PDCInit

Initializes the connection to the PDC.

Prototype:

```
void
PDCInit(void)
```

Description:

This function will enable clocking to the SSI and GPIO A modules, configure the GPIO pins to be used for an SSI interface, and it will configure the SSI as a 1 Mbps master device, operating in MOTO mode. It will also enable the SSI module, and will enable the chip select for the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

34.2.2.7 PDCLCDBacklightOff

Turn off the backlight.

Prototype:

```
void
PDCLCDBacklightOff(void)
```

Description:

This function turns off the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

34.2.2.8 PDCLCDBacklightOn

Turns on the backlight.

Prototype:

```
void
PDCLCDBacklightOn(void)
```

Description:

This function turns on the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

34.2.2.9 PDCLCDClear

Clear the screen.

Prototype:

```
void
PDCLCDClear(void)
```

Description:

This function clears the contents of the LCD screen. The cursor will be returned to the upper left corner.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

34.2.2.10 PDCLCDCreateChar

Write a character pattern to the LCD.

Prototype:

```
void
PDCLCDCreateChar(unsigned char ucChar,
                 unsigned char *pucData)
```

Parameters:

ucChar is the character index to create. Valid values are zero through seven.

pucData is the data for the character pattern. It contains eight bytes, with the first byte being the top row of the pattern. In each byte, the LSB is the right pixel of the pattern.

Description:

This function will write a character pattern into the LCD for use as a character to be displayed. After writing the pattern, it can be used on the LCD by writing the corresponding character index to the display.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

34.2.2.11 PDCLCDInit

Initializes the LCD display.

Prototype:
```
void
PDCLCDInit(void)
```

Description:
This function will set up the LCD display for writing. It will set the data bus to 8 bits, set the number of lines to 2, and the font size to 5x10. It will also turn the display off, clear the display, turn the display back on, and enable the backlight.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Note:
The PDC must be initialized via the PDCInit() function before this function can be called. Also, it may be necessary to adjust the contrast potentiometer in order to discern any output on the LCD display.

Returns:
None.

34.2.2.12 PDCLCDSetPos

Set the position of the cursor.

Prototype:
```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

Parameters:
ucX is the horizontal position. Valid values are zero through fifteen.
ucY is the vertical position.. Valid values are zero and one.

Description:
This function will move the cursor to the specified position. All characters written to the LCD are placed at the current cursor position, which is automatically advanced.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

34.2.2.13 PDCLCDWrite

Writes a string to the LCD display.

Prototype:

```
void
PDCLCDWrite(const char *pcStr,
            unsigned long ulCount)
```

Parameters:

pcStr pointer to the string to be displayed.

ulCount is the number of characters to be displayed.

Description:

This function will display a string on the LCD at the current cursor position. It is the caller's responsibility to position the cursor to the place where the string should be displayed (either explicitly via PDCLCDSetPos() or implicitly from where the cursor was left after a previous call to PDCLCDWrite()), and to properly account for the LCD boundary (line wrapping is not automatically performed). Null characters are not treated special and are written to the LCD, which interprets it as a special programmable character glyph (see PDCLCDCreateChar()).

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

34.2.2.14 PDCLEDRead

Read the current status of the PDC LEDs.

Prototype:

```
unsigned char
PDCLEDRead(void)
```

Description:

This function will read the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The value currently displayed by the LEDs.

34.2.2.15 PDCLEDWrite

Write to the PDC LEDs.

Prototype:

```
void
PDCLEDWrite(unsigned char ucLED)
```

Parameters:

ucLED value to write to the LEDs.

Description:

This function set the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

34.2.2.16 PDCRead

Read a PDC register.

Prototype:

```
unsigned char
PDCRead(unsigned char ucAddr)
```

Parameters:

ucAddr specifies the PDC register to read.

Description:

This function will perform the SSI transfers required to read a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

Returns the value read from the PDC.

34.2.2.17 PDCWrite

Write a PDC register.

Prototype:

```
void
PDCWrite(unsigned char ucAddr,
         unsigned char ucData)
```

Parameters:

ucAddr specifies the PDC register to write.

ucData specifies the data to write.

Description:

This function will perform the SSI transfers required to write a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

34.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

Comparator (comparator)

This example application demonstrates the operation of the analog comparator(s). Comparator zero (which is present on all devices that have analog comparators) is configured to compare its negative input to an internally generated 1.65 V reference and toggle the state of the LED on port B0 based on comparator change interrupts. The LED will be turned on by the interrupt handler when a rising edge on the comparator output is detected, and will be turned off when a falling edge is detected.

In order for this example to work properly, the ULED0 (JP22) jumper must be installed on the board.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 of Sandstorm-class Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata is fixed in Rev C2 of Sandstorm-class Stellaris microcontrollers.

GPIO (gpio_led)

This example application uses LEDs connected to GPIO pins to create a “roving eye” display. Port B0-B3 are driven in a sequential manner to give the illusion of an eye looking back and forth.

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), ULED2 (JP24), and ULED3 (JP25) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the LCD and is a starting point for more complicated applications.

I2C (i2c_atmel)

This example application uses the I2C master to communicate with the Atmel AT24C08A EEPROM that is on the development board. The first sixteen bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the I2C interrupt; since a sixteen-byte read at a 100 kHz I2C bus speed takes almost 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

In order for this example to work properly, the I2C_SCL (JP14), I2C_SDA (JP13), and I2CM_A2 (JP11) jumpers must be installed on the board, and the I2CM_WP (JP12) jumper must be removed.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; individual LEDs connected to port B0-B2 will be turned on upon interrupt handler entry and off before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), and ULED2 (JP24) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 50 kHz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

DK-LM3S801 Quickstart Application (qs_dk-lm3s801)

This example uses the potentiometer on the development board to vary the rate and frequency of a repetitive beep from the piezo buzzer. Turning the knob in one direction will result in slower beeps at lower frequency, while turning it the other direction will result in faster beeps at a higher frequency. The potentiometer setting along with the tone "note" is displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn the beeping noise on and off; when off the LCD and UART still show the setting.

SSI (ssi_atmel)

This example application uses the SSI master to communicate with the Atmel AT25F1024A EEPROM that is on the development board. The first 256 bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the SSI interrupt; since a 256-byte read at a 1 MHz SSI bus speed takes around 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own GPIO (port B0 and B1) on each interrupt; the attached LED will indicate the occurrence and rate of interrupts.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (the SER0 connector on the Stellaris Family Development Board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port B0 is inverted so that it is easy to see that it is being fed, which occurs once every second.

35 DK-LM3S811 Example Applications

35.1 Introduction

The DK-LM3S811 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the Peripheral Device Controller on the Stellaris Family Development Kit board. The PDC is used to access the character LCD, eight user LEDs, eight user DIP switches, and twenty-four GPIOs.

There is an IAR workspace file (`dk-lm3s811.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`dk-lm3s811-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`dk-lm3s811.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/dk-lm3s811` subdirectory of the peripheral driver library source distribution.

35.2 API Functions

Functions

- unsigned char PDCDIPRead (void)
- unsigned char PDCGPIODirRead (unsigned char ucIdx)
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)
- unsigned char PDCGPIORead (unsigned char ucIdx)
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)
- void PDCInit (void)
- void PDCLCDBacklightOff (void)
- void PDCLCDBacklightOn (void)
- void PDCLCDClear (void)
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)
- void PDCLCDInit (void)
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)

- unsigned char PDCLEDRead (void)
- void PDCLEDWrite (unsigned char ucLED)
- unsigned char PDCRead (unsigned char ucAddr)
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)

35.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

35.2.2 Function Documentation

35.2.2.1 PDCDIPRead

Read the current value of the PDC DIP switches.

Prototype:

```
unsigned char
PDCDIPRead(void)
```

Description:

This function will read the current value of the DIP switches attached to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The current state of the DIP switches.

35.2.2.2 PDCGPIODirRead

Reads a GPIO direction register.

Prototype:

```
unsigned char
PDCGPIODirRead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO direction registers in the PDC. The direction bit is set for pins that are outputs and clear for pins that are inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the direction register.

35.2.2.3 PDCGPIODirWrite

Write a GPIO direction register.

Prototype:

```
void
PDCGPIODirWrite(unsigned char ucIdx,
                unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO direction register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO direction register.

Description:

This function writes ones of the GPIO direction registers in the PDC. The direction bit should be set for pins that are to be outputs and clear for pins that are to be inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

35.2.2.4 PDCGPIORead

Reads a GPIO data register.

Prototype:

```
unsigned char
PDCGPIORead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO data registers in the PDC. The value returned for a pin is the value being driven out for outputs or the value being read for inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the data register.

35.2.2.5 PDCGPIOWrite

Write a GPIO data register.

Prototype:

```
void
PDCGPIOWrite(unsigned char ucIdx,
             unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO data register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO data register.

Description:

This function writes one of the GPIO direction registers in the PDC. The written to a pin is driven out for output pins and ignored for input pins.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

35.2.2.6 PDCInit

Initializes the connection to the PDC.

Prototype:

```
void
PDCInit(void)
```

Description:

This function will enable clocking to the SSI and GPIO A modules, configure the GPIO pins to be used for an SSI interface, and it will configure the SSI as a 1 Mbps master device, operating in MOTO mode. It will also enable the SSI module, and will enable the chip select for the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

35.2.2.7 PDCLCDBacklightOff

Turn off the backlight.

Prototype:

```
void
PDCLCDBacklightOff(void)
```

Description:

This function turns off the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

35.2.2.8 PDCLCDBacklightOn

Turns on the backlight.

Prototype:

```
void
PDCLCDBacklightOn(void)
```

Description:

This function turns on the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

35.2.2.9 PDCLCDClear

Clear the screen.

Prototype:

```
void
PDCLCDClear(void)
```

Description:

This function clears the contents of the LCD screen. The cursor will be returned to the upper left corner.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

35.2.2.10 PDCLCDCreateChar

Write a character pattern to the LCD.

Prototype:

```
void
PDCLCDCreateChar(unsigned char ucChar,
                 unsigned char *pucData)
```

Parameters:

ucChar is the character index to create. Valid values are zero through seven.

pucData is the data for the character pattern. It contains eight bytes, with the first byte being the top row of the pattern. In each byte, the LSB is the right pixel of the pattern.

Description:

This function will write a character pattern into the LCD for use as a character to be displayed. After writing the pattern, it can be used on the LCD by writing the corresponding character index to the display.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

35.2.2.11 PDCLCDInit

Initializes the LCD display.

Prototype:
```
void
PDCLCDInit(void)
```

Description:
This function will set up the LCD display for writing. It will set the data bus to 8 bits, set the number of lines to 2, and the font size to 5x10. It will also turn the display off, clear the display, turn the display back on, and enable the backlight.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Note:
The PDC must be initialized via the PDCInit() function before this function can be called. Also, it may be necessary to adjust the contrast potentiometer in order to discern any output on the LCD display.

Returns:
None.

35.2.2.12 PDCLCDSetPos

Set the position of the cursor.

Prototype:
```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

Parameters:
ucX is the horizontal position. Valid values are zero through fifteen.
ucY is the vertical position.. Valid values are zero and one.

Description:
This function will move the cursor to the specified position. All characters written to the LCD are placed at the current cursor position, which is automatically advanced.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

35.2.2.13 PDCLCDWrite

Writes a string to the LCD display.

Prototype:

```
void
PDCLCDWrite(const char *pcStr,
            unsigned long ulCount)
```

Parameters:

pcStr pointer to the string to be displayed.

ulCount is the number of characters to be displayed.

Description:

This function will display a string on the LCD at the current cursor position. It is the caller's responsibility to position the cursor to the place where the string should be displayed (either explicitly via PDCLCDSetPos() or implicitly from where the cursor was left after a previous call to PDCLCDWrite()), and to properly account for the LCD boundary (line wrapping is not automatically performed). Null characters are not treated special and are written to the LCD, which interprets it as a special programmable character glyph (see PDCLCDCreateChar()).

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

35.2.2.14 PDCLEDRead

Read the current status of the PDC LEDs.

Prototype:

```
unsigned char
PDCLEDRead(void)
```

Description:

This function will read the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The value currently displayed by the LEDs.

35.2.2.15 PDCLEDWrite

Write to the PDC LEDs.

Prototype:

```
void
PDCLEDWrite(unsigned char ucLED)
```

Parameters:

ucLED value to write to the LEDs.

Description:

This function set the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

35.2.2.16 PDCRead

Read a PDC register.

Prototype:

```
unsigned char
PDCRead(unsigned char ucAddr)
```

Parameters:

ucAddr specifies the PDC register to read.

Description:

This function will perform the SSI transfers required to read a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

Returns the value read from the PDC.

35.2.2.17 PDCWrite

Write a PDC register.

Prototype:

```
void
PDCWrite(unsigned char ucAddr,
         unsigned char ucData)
```

Parameters:

ucAddr specifies the PDC register to write.

ucData specifies the data to write.

Description:

This function will perform the SSI transfers required to write a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

35.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

Comparator (comparator)

This example application demonstrates the operation of the analog comparator(s). Comparator zero (which is present on all devices that have analog comparators) is configured to compare its negative input to an internally generated 1.65 V reference and toggle the state of the LED on port B0 based on comparator change interrupts. The LED will be turned on by the interrupt handler when a rising edge on the comparator output is detected, and will be turned off when a falling edge is detected.

In order for this example to work properly, the ULED0 (JP22) jumper must be installed on the board.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 of Sandstorm-class Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata is fixed in Rev C2 of Sandstorm-class Stellaris microcontrollers.

GPIO (gpio_led)

This example application uses LEDs connected to GPIO pins to create a “roving eye” display. Port B0-B3 are driven in a sequential manner to give the illusion of an eye looking back and forth.

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), ULED2 (JP24), and ULED3 (JP25) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the LCD and is a starting point for more complicated applications.

I2C (i2c_atmel)

This example application uses the I2C master to communicate with the Atmel AT24C08A EEPROM that is on the development board. The first sixteen bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the I2C interrupt; since a sixteen-byte read at a 100 kHz I2C bus speed takes almost 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

In order for this example to work properly, the I2C_SCL (JP14), I2C_SDA (JP13), and I2CM_A2 (JP11) jumpers must be installed on the board, and the I2CM_WP (JP12) jumper must be removed.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; individual LEDs connected to port B0-B2 will be turned on upon interrupt handler entry and off before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), and ULED2 (JP24) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 50 kHz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

DK-LM3S811 Quickstart Application (qs_dk-lm3s811)

This example uses the potentiometer on the development board to vary the rate of a repetitive beep from the piezo buzzer, while the light sensor will vary the frequency of the beep. Turning the knob in one direction will result in slower beeps while turning it in the other direction will result in faster beeps. The amount of light falling on the light sensor affects the frequency of the beep. The more light falling on the sensor the higher the pitch of the beep. The potentiometer setting along with the "note" representing the pitch of the beep is displayed on the LCD, and a log of the readings is

output on the UART at 115,200, 8-n-1. The push button can be used to turn the beeping noise on and off; when off the LCD and UART still provide the settings.

In the default jumper configuration of the development board, the push button will not actually mute the beep. In order for this example to fully work, the following jumper wire connections must be made: JP19 pin 2 to J6 pin 6.

SSI (ssi_atmel)

This example application uses the SSI master to communicate with the Atmel AT25F1024A EEPROM that is on the development board. The first 256 bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the SSI interrupt; since a 256-byte read at a 1 MHz SSI bus speed takes around 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own GPIO (port B0 and B1) on each interrupt; the attached LED will indicate the occurrence and rate of interrupts.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (the SER0 connector on the Stellaris Family Development Board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port B0 is inverted so that it is easy to see that it is being fed, which occurs once every second.

36 DK-LM3S815 Example Applications

36.1 Introduction

The DK-LM3S815 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the Peripheral Device Controller on the Stellaris Family Development Kit board. The PDC is used to access the character LCD, eight user LEDs, eight user DIP switches, and twenty-four GPIOs.

There is an IAR workspace file (`dk-lm3s815.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`dk-lm3s815-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`dk-lm3s815.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/dk-lm3s815` subdirectory of the peripheral driver library source distribution.

36.2 API Functions

Functions

- unsigned char PDCDIPRead (void)
- unsigned char PDCGPIODirRead (unsigned char ucIdx)
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)
- unsigned char PDCGPIORead (unsigned char ucIdx)
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)
- void PDCInit (void)
- void PDCLCDBacklightOff (void)
- void PDCLCDBacklightOn (void)
- void PDCLCDClear (void)
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)
- void PDCLCDInit (void)
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)

- unsigned char PDCLEDRead (void)
- void PDCLEDWrite (unsigned char ucLED)
- unsigned char PDCRead (unsigned char ucAddr)
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)

36.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

36.2.2 Function Documentation

36.2.2.1 PDCDIPRead

Read the current value of the PDC DIP switches.

Prototype:

```
unsigned char
PDCDIPRead(void)
```

Description:

This function will read the current value of the DIP switches attached to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The current state of the DIP switches.

36.2.2.2 PDCGPIODirRead

Reads a GPIO direction register.

Prototype:

```
unsigned char
PDCGPIODirRead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO direction registers in the PDC. The direction bit is set for pins that are outputs and clear for pins that are inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the direction register.

36.2.2.3 PDCGPIODirWrite

Write a GPIO direction register.

Prototype:

```
void
PDCGPIODirWrite(unsigned char ucIdx,
                unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO direction register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO direction register.

Description:

This function writes ones of the GPIO direction registers in the PDC. The direction bit should be set for pins that are to be outputs and clear for pins that are to be inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

36.2.2.4 PDCGPIORead

Reads a GPIO data register.

Prototype:

```
unsigned char
PDCGPIORead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO data registers in the PDC. The value returned for a pin is the value being driven out for outputs or the value being read for inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the data register.

36.2.2.5 PDCGPIOWrite

Write a GPIO data register.

Prototype:

```
void
PDCGPIOWrite(unsigned char ucIdx,
             unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO data register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO data register.

Description:

This function writes one of the GPIO direction registers in the PDC. The written to a pin is driven out for output pins and ignored for input pins.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

36.2.2.6 PDCInit

Initializes the connection to the PDC.

Prototype:

```
void
PDCInit(void)
```

Description:

This function will enable clocking to the SSI and GPIO A modules, configure the GPIO pins to be used for an SSI interface, and it will configure the SSI as a 1 Mbps master device, operating in MOTO mode. It will also enable the SSI module, and will enable the chip select for the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

36.2.2.7 PDCLCDBacklightOff

Turn off the backlight.

Prototype:

```
void
PDCLCDBacklightOff(void)
```

Description:

This function turns off the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

36.2.2.8 PDCLCDBacklightOn

Turns on the backlight.

Prototype:

```
void
PDCLCDBacklightOn(void)
```

Description:

This function turns on the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

36.2.2.9 PDCLCDClear

Clear the screen.

Prototype:

```
void
PDCLCDClear(void)
```

Description:

This function clears the contents of the LCD screen. The cursor will be returned to the upper left corner.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

36.2.2.10 PDCLCDCreateChar

Write a character pattern to the LCD.

Prototype:

```
void
PDCLCDCreateChar(unsigned char ucChar,
                 unsigned char *pucData)
```

Parameters:

ucChar is the character index to create. Valid values are zero through seven.

pucData is the data for the character pattern. It contains eight bytes, with the first byte being the top row of the pattern. In each byte, the LSB is the right pixel of the pattern.

Description:

This function will write a character pattern into the LCD for use as a character to be displayed. After writing the pattern, it can be used on the LCD by writing the corresponding character index to the display.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

36.2.2.11 PDCLCDInit

Initializes the LCD display.

Prototype:
```
void
PDCLCDInit(void)
```

Description:
This function will set up the LCD display for writing. It will set the data bus to 8 bits, set the number of lines to 2, and the font size to 5x10. It will also turn the display off, clear the display, turn the display back on, and enable the backlight.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Note:
The PDC must be initialized via the PDCInit() function before this function can be called. Also, it may be necessary to adjust the contrast potentiometer in order to discern any output on the LCD display.

Returns:
None.

36.2.2.12 PDCLCDSetPos

Set the position of the cursor.

Prototype:
```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

Parameters:
ucX is the horizontal position. Valid values are zero through fifteen.
ucY is the vertical position.. Valid values are zero and one.

Description:
This function will move the cursor to the specified position. All characters written to the LCD are placed at the current cursor position, which is automatically advanced.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

36.2.2.13 PDCLCDWrite

Writes a string to the LCD display.

Prototype:

```
void
PDCLCDWrite(const char *pcStr,
            unsigned long ulCount)
```

Parameters:

pcStr pointer to the string to be displayed.

ulCount is the number of characters to be displayed.

Description:

This function will display a string on the LCD at the current cursor position. It is the caller's responsibility to position the cursor to the place where the string should be displayed (either explicitly via PDCLCDSetPos() or implicitly from where the cursor was left after a previous call to PDCLCDWrite()), and to properly account for the LCD boundary (line wrapping is not automatically performed). Null characters are not treated special and are written to the LCD, which interprets it as a special programmable character glyph (see PDCLCDCreateChar()).

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

36.2.2.14 PDCLEDRead

Read the current status of the PDC LEDs.

Prototype:

```
unsigned char
PDCLEDRead(void)
```

Description:

This function will read the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The value currently displayed by the LEDs.

36.2.2.15 PDCLEDWrite

Write to the PDC LEDs.

Prototype:

```
void
PDCLEDWrite(unsigned char ucLED)
```

Parameters:

ucLED value to write to the LEDs.

Description:

This function set the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

36.2.2.16 PDCRead

Read a PDC register.

Prototype:

```
unsigned char
PDCRead(unsigned char ucAddr)
```

Parameters:

ucAddr specifies the PDC register to read.

Description:

This function will perform the SSI transfers required to read a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

Returns the value read from the PDC.

36.2.2.17 PDCWrite

Write a PDC register.

Prototype:

```
void
PDCWrite(unsigned char ucAddr,
         unsigned char ucData)
```

Parameters:

ucAddr specifies the PDC register to write.

ucData specifies the data to write.

Description:

This function will perform the SSI transfers required to write a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

36.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

Comparator (comparator)

This example application demonstrates the operation of the analog comparator(s). Comparator zero (which is present on all devices that have analog comparators) is configured to compare its negative input to an internally generated 1.65 V reference and toggle the state of the LED on port B0 based on comparator change interrupts. The LED will be turned on by the interrupt handler when a rising edge on the comparator output is detected, and will be turned off when a falling edge is detected.

In order for this example to work properly, the ULED0 (JP22) jumper must be installed on the board.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 of Sandstorm-class Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata is fixed in Rev C2 of Sandstorm-class Stellaris microcontrollers.

GPIO (gpio_led)

This example application uses LEDs connected to GPIO pins to create a “roving eye” display. Port B0-B3 are driven in a sequential manner to give the illusion of an eye looking back and forth.

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), ULED2 (JP24), and ULED3 (JP25) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the LCD and is a starting point for more complicated applications.

I2C (i2c_atmel)

This example application uses the I2C master to communicate with the Atmel AT24C08A EEPROM that is on the development board. The first sixteen bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the I2C interrupt; since a sixteen-byte read at a 100 kHz I2C bus speed takes almost 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

In order for this example to work properly, the I2C_SCL (JP14), I2C_SDA (JP13), and I2CM_A2 (JP11) jumpers must be installed on the board, and the I2CM_WP (JP12) jumper must be removed.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; individual LEDs connected to port B0-B2 will be turned on upon interrupt handler entry and off before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), and ULED2 (JP24) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 50 kHz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

DK-LM3S815 Quickstart Application (qs_dk-lm3s815)

This example uses the potentiometer on the development board to vary the rate of a repetitive beep from the piezo buzzer, while the light sensor will vary the frequency of the beep. Turning the knob in one direction will result in slower beeps while turning it in the other direction will result in faster beeps. The amount of light falling on the light sensor affects the frequency of the beep. The more light falling on the sensor the higher the pitch of the beep. The potentiometer setting along with the "note" representing the pitch of the beep is displayed on the LCD, and a log of the readings is

output on the UART at 115,200, 8-n-1. The push button can be used to turn the beeping noise on and off; when off the LCD and UART still provide the settings.

In the default jumper configuration of the development board, the push button will not actually mute the beep. In order for this example to fully work, the following jumper wire connections must be made: JP19 pin 2 to J6 pin 6.

SSI (ssi_atmel)

This example application uses the SSI master to communicate with the Atmel AT25F1024A EEPROM that is on the development board. The first 256 bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the SSI interrupt; since a 256-byte read at a 1 MHz SSI bus speed takes around 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own GPIO (port B0 and B1) on each interrupt; the attached LED will indicate the occurrence and rate of interrupts.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (the SER0 connector on the Stellaris Family Development Board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port B0 is inverted so that it is easy to see that it is being fed, which occurs once every second.

37 DK-LM3S817 Example Applications

37.1 Introduction

The DK-LM3S817 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the Peripheral Device Controller on the Stellaris Family Development Kit board. The PDC is used to access the character LCD, eight user LEDs, eight user DIP switches, and twenty-four GPIOs.

There is an IAR workspace file (`dk-lm3s817.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`dk-lm3s817-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`dk-lm3s817.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/dk-lm3s817` subdirectory of the peripheral driver library source distribution.

37.2 API Functions

Functions

- unsigned char PDCDIPRead (void)
- unsigned char PDCGPIODirRead (unsigned char ucIdx)
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)
- unsigned char PDCGPIORead (unsigned char ucIdx)
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)
- void PDCInit (void)
- void PDCLCDBacklightOff (void)
- void PDCLCDBacklightOn (void)
- void PDCLCDClear (void)
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)
- void PDCLCDInit (void)
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)

- unsigned char PDCLEDRead (void)
- void PDCLEDWrite (unsigned char ucLED)
- unsigned char PDCRead (unsigned char ucAddr)
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)

37.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

37.2.2 Function Documentation

37.2.2.1 PDCDIPRead

Read the current value of the PDC DIP switches.

Prototype:

```
unsigned char
PDCDIPRead(void)
```

Description:

This function will read the current value of the DIP switches attached to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The current state of the DIP switches.

37.2.2.2 PDCGPIODirRead

Reads a GPIO direction register.

Prototype:

```
unsigned char
PDCGPIODirRead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO direction registers in the PDC. The direction bit is set for pins that are outputs and clear for pins that are inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the direction register.

37.2.2.3 PDCGPIODirWrite

Write a GPIO direction register.

Prototype:

```
void
PDCGPIODirWrite(unsigned char ucIdx,
                unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO direction register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO direction register.

Description:

This function writes ones of the GPIO direction registers in the PDC. The direction bit should be set for pins that are to be outputs and clear for pins that are to be inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

37.2.2.4 PDCGPIORead

Reads a GPIO data register.

Prototype:

```
unsigned char
PDCGPIORead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO data registers in the PDC. The value returned for a pin is the value being driven out for outputs or the value being read for inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the data register.

37.2.2.5 PDCGPIOWrite

Write a GPIO data register.

Prototype:

```
void
PDCGPIOWrite(unsigned char ucIdx,
             unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO data register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO data register.

Description:

This function writes one of the GPIO direction registers in the PDC. The written to a pin is driven out for output pins and ignored for input pins.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

37.2.2.6 PDCInit

Initializes the connection to the PDC.

Prototype:

```
void
PDCInit(void)
```

Description:

This function will enable clocking to the SSI and GPIO A modules, configure the GPIO pins to be used for an SSI interface, and it will configure the SSI as a 1 Mbps master device, operating in MOTO mode. It will also enable the SSI module, and will enable the chip select for the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

37.2.2.7 PDCLCDBacklightOff

Turn off the backlight.

Prototype:

```
void
PDCLCDBacklightOff(void)
```

Description:

This function turns off the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

37.2.2.8 PDCLCDBacklightOn

Turns on the backlight.

Prototype:

```
void
PDCLCDBacklightOn(void)
```

Description:

This function turns on the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

37.2.2.9 PDCLCDClear

Clear the screen.

Prototype:

```
void
PDCLCDClear(void)
```

Description:

This function clears the contents of the LCD screen. The cursor will be returned to the upper left corner.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

37.2.2.10 PDCLCDCreateChar

Write a character pattern to the LCD.

Prototype:

```
void
PDCLCDCreateChar(unsigned char ucChar,
                 unsigned char *pucData)
```

Parameters:

ucChar is the character index to create. Valid values are zero through seven.

pucData is the data for the character pattern. It contains eight bytes, with the first byte being the top row of the pattern. In each byte, the LSB is the right pixel of the pattern.

Description:

This function will write a character pattern into the LCD for use as a character to be displayed. After writing the pattern, it can be used on the LCD by writing the corresponding character index to the display.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

37.2.2.11 PDCLCDInit

Initializes the LCD display.

Prototype:
```
void
PDCLCDInit(void)
```

Description:
This function will set up the LCD display for writing. It will set the data bus to 8 bits, set the number of lines to 2, and the font size to 5x10. It will also turn the display off, clear the display, turn the display back on, and enable the backlight.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Note:
The PDC must be initialized via the PDCInit() function before this function can be called. Also, it may be necessary to adjust the contrast potentiometer in order to discern any output on the LCD display.

Returns:
None.

37.2.2.12 PDCLCDSetPos

Set the position of the cursor.

Prototype:
```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

Parameters:
ucX is the horizontal position. Valid values are zero through fifteen.
ucY is the vertical position.. Valid values are zero and one.

Description:
This function will move the cursor to the specified position. All characters written to the LCD are placed at the current cursor position, which is automatically advanced.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

37.2.2.13 PDCLCDWrite

Writes a string to the LCD display.

Prototype:

```
void
PDCLCDWrite(const char *pcStr,
            unsigned long ulCount)
```

Parameters:

pcStr pointer to the string to be displayed.

ulCount is the number of characters to be displayed.

Description:

This function will display a string on the LCD at the current cursor position. It is the caller's responsibility to position the cursor to the place where the string should be displayed (either explicitly via PDCLCDSetPos() or implicitly from where the cursor was left after a previous call to PDCLCDWrite()), and to properly account for the LCD boundary (line wrapping is not automatically performed). Null characters are not treated special and are written to the LCD, which interprets it as a special programmable character glyph (see PDCLCDCreateChar()).

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

37.2.2.14 PDCLEDRead

Read the current status of the PDC LEDs.

Prototype:

```
unsigned char
PDCLEDRead(void)
```

Description:

This function will read the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The value currently displayed by the LEDs.

37.2.2.15 PDCLEDWrite

Write to the PDC LEDs.

Prototype:

```
void
PDCLEDWrite(unsigned char ucLED)
```

Parameters:

ucLED value to write to the LEDs.

Description:

This function set the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

37.2.2.16 PDCRead

Read a PDC register.

Prototype:

```
unsigned char
PDCRead(unsigned char ucAddr)
```

Parameters:

ucAddr specifies the PDC register to read.

Description:

This function will perform the SSI transfers required to read a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

Returns the value read from the PDC.

37.2.2.17 PDCWrite

Write a PDC register.

Prototype:

```
void
PDCWrite(unsigned char ucAddr,
         unsigned char ucData)
```

Parameters:

ucAddr specifies the PDC register to write.

ucData specifies the data to write.

Description:

This function will perform the SSI transfers required to write a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

37.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

Comparator (comparator)

This example application demonstrates the operation of the analog comparator(s). Comparator zero (which is present on all devices that have analog comparators) is configured to compare its negative input to an internally generated 1.65 V reference and toggle the state of the LED on port B0 based on comparator change interrupts. The LED will be turned on by the interrupt handler when a rising edge on the comparator output is detected, and will be turned off when a falling edge is detected.

In order for this example to work properly, the ULED0 (JP22) jumper must be installed on the board.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 of Sandstorm-class Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata is fixed in Rev C2 of Sandstorm-class Stellaris microcontrollers.

GPIO (gpio_led)

This example application uses LEDs connected to GPIO pins to create a “roving eye” display. Port B0-B3 are driven in a sequential manner to give the illusion of an eye looking back and forth.

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), ULED2 (JP24), and ULED3 (JP25) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the LCD and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; individual LEDs connected to port B0-B2 will be turned on upon interrupt handler entry and off before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), and ULED2 (JP24) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 50 kHz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

DK-LM3S817 Quickstart Application (qs_dk-lm3s817)

This example uses the potentiometer on the development board to vary the rate of a repetitive beep from the piezo buzzer, while the light sensor will vary the frequency of the beep. Turning the knob in one direction will result in slower beeps while turning it in the other direction will result in faster beeps. The amount of light falling on the light sensor affects the frequency of the beep. The more light falling on the sensor the higher the pitch of the beep. The potentiometer setting along with the "note" representing the pitch of the beep is displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn the beeping noise on and off; when off the LCD and UART still provide the settings.

In the default jumper configuration of the development board, the push button will not actually mute the beep. In order for this example to fully work, the following jumper wire connections must be made: JP19 pin 2 to J6 pin 6.

SSI (ssi_atmel)

This example application uses the SSI master to communicate with the Atmel AT25F1024A EEPROM that is on the development board. The first 256 bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the SSI interrupt; since a 256-byte

read at a 1 MHz SSI bus speed takes around 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own GPIO (port B0 and B1) on each interrupt; the attached LED will indicate the occurrence and rate of interrupts.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (the SER0 connector on the Stellaris Family Development Board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port B0 is inverted so that it is easy to see that it is being fed, which occurs once every second.

38 DK-LM3S818 Example Applications

38.1 Introduction

The DK-LM3S818 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the Peripheral Device Controller on the Stellaris Family Development Kit board. The PDC is used to access the character LCD, eight user LEDs, eight user DIP switches, and twenty-four GPIOs.

There is an IAR workspace file (`dk-lm3s818.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`dk-lm3s818-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`dk-lm3s818.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/dk-lm3s818` subdirectory of the peripheral driver library source distribution.

38.2 API Functions

Functions

- unsigned char PDCDIPRead (void)
- unsigned char PDCGPIODirRead (unsigned char ucIdx)
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)
- unsigned char PDCGPIORead (unsigned char ucIdx)
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)
- void PDCInit (void)
- void PDCLCDBacklightOff (void)
- void PDCLCDBacklightOn (void)
- void PDCLCDClear (void)
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)
- void PDCLCDInit (void)
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)

- unsigned char PDCLEDRead (void)
- void PDCLEDWrite (unsigned char ucLED)
- unsigned char PDCRead (unsigned char ucAddr)
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)

38.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

38.2.2 Function Documentation

38.2.2.1 PDCDIPRead

Read the current value of the PDC DIP switches.

Prototype:

```
unsigned char
PDCDIPRead(void)
```

Description:

This function will read the current value of the DIP switches attached to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The current state of the DIP switches.

38.2.2.2 PDCGPIODirRead

Reads a GPIO direction register.

Prototype:

```
unsigned char
PDCGPIODirRead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO direction registers in the PDC. The direction bit is set for pins that are outputs and clear for pins that are inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the direction register.

38.2.2.3 PDCGPIODirWrite

Write a GPIO direction register.

Prototype:

```
void
PDCGPIODirWrite(unsigned char ucIdx,
                unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO direction register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO direction register.

Description:

This function writes ones of the GPIO direction registers in the PDC. The direction bit should be set for pins that are to be outputs and clear for pins that are to be inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

38.2.2.4 PDCGPIORead

Reads a GPIO data register.

Prototype:

```
unsigned char
PDCGPIORead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO data registers in the PDC. The value returned for a pin is the value being driven out for outputs or the value being read for inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the data register.

38.2.2.5 PDCGPIOWrite

Write a GPIO data register.

Prototype:

```
void
PDCGPIOWrite(unsigned char ucIdx,
             unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO data register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO data register.

Description:

This function writes one of the GPIO direction registers in the PDC. The written to a pin is driven out for output pins and ignored for input pins.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

38.2.2.6 PDCInit

Initializes the connection to the PDC.

Prototype:

```
void
PDCInit(void)
```

Description:

This function will enable clocking to the SSI and GPIO A modules, configure the GPIO pins to be used for an SSI interface, and it will configure the SSI as a 1 Mbps master device, operating in MOTO mode. It will also enable the SSI module, and will enable the chip select for the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

38.2.2.7 PDCLCDBacklightOff

Turn off the backlight.

Prototype:

```
void
PDCLCDBacklightOff(void)
```

Description:

This function turns off the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

38.2.2.8 PDCLCDBacklightOn

Turns on the backlight.

Prototype:

```
void
PDCLCDBacklightOn(void)
```

Description:

This function turns on the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

38.2.2.9 PDCLCDClear

Clear the screen.

Prototype:

```
void
PDCLCDClear(void)
```

Description:

This function clears the contents of the LCD screen. The cursor will be returned to the upper left corner.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

38.2.2.10 PDCLCDCreateChar

Write a character pattern to the LCD.

Prototype:

```
void
PDCLCDCreateChar(unsigned char ucChar,
                 unsigned char *pucData)
```

Parameters:

ucChar is the character index to create. Valid values are zero through seven.

pucData is the data for the character pattern. It contains eight bytes, with the first byte being the top row of the pattern. In each byte, the LSB is the right pixel of the pattern.

Description:

This function will write a character pattern into the LCD for use as a character to be displayed. After writing the pattern, it can be used on the LCD by writing the corresponding character index to the display.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

38.2.2.11 PDCLCDInit

Initializes the LCD display.

Prototype:
```
void
PDCLCDInit(void)
```

Description:
This function will set up the LCD display for writing. It will set the data bus to 8 bits, set the number of lines to 2, and the font size to 5x10. It will also turn the display off, clear the display, turn the display back on, and enable the backlight.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Note:
The PDC must be initialized via the PDCInit() function before this function can be called. Also, it may be necessary to adjust the contrast potentiometer in order to discern any output on the LCD display.

Returns:
None.

38.2.2.12 PDCLCDSetPos

Set the position of the cursor.

Prototype:
```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

Parameters:
ucX is the horizontal position. Valid values are zero through fifteen.
ucY is the vertical position.. Valid values are zero and one.

Description:
This function will move the cursor to the specified position. All characters written to the LCD are placed at the current cursor position, which is automatically advanced.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

38.2.2.13 PDCLCDWrite

Writes a string to the LCD display.

Prototype:

```
void
PDCLCDWrite(const char *pcStr,
            unsigned long ulCount)
```

Parameters:

pcStr pointer to the string to be displayed.

ulCount is the number of characters to be displayed.

Description:

This function will display a string on the LCD at the current cursor position. It is the caller's responsibility to position the cursor to the place where the string should be displayed (either explicitly via PDCLCDSetPos() or implicitly from where the cursor was left after a previous call to PDCLCDWrite()), and to properly account for the LCD boundary (line wrapping is not automatically performed). Null characters are not treated special and are written to the LCD, which interprets it as a special programmable character glyph (see PDCLCDCreateChar()).

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

38.2.2.14 PDCLEDRead

Read the current status of the PDC LEDs.

Prototype:

```
unsigned char
PDCLEDRead(void)
```

Description:

This function will read the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The value currently displayed by the LEDs.

38.2.2.15 PDCLEDWrite

Write to the PDC LEDs.

Prototype:

```
void
PDCLEDWrite(unsigned char ucLED)
```

Parameters:

ucLED value to write to the LEDs.

Description:

This function set the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

38.2.2.16 PDCRead

Read a PDC register.

Prototype:

```
unsigned char
PDCRead(unsigned char ucAddr)
```

Parameters:

ucAddr specifies the PDC register to read.

Description:

This function will perform the SSI transfers required to read a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

Returns the value read from the PDC.

38.2.2.17 PDCWrite

Write a PDC register.

Prototype:

```
void
PDCWrite(unsigned char ucAddr,
         unsigned char ucData)
```

Parameters:

ucAddr specifies the PDC register to write.

ucData specifies the data to write.

Description:

This function will perform the SSI transfers required to write a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

38.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

Comparator (comparator)

This example application demonstrates the operation of the analog comparator(s). Comparator zero (which is present on all devices that have analog comparators) is configured to compare its negative input to an internally generated 1.65 V reference and toggle the state of the LED on port B0 based on comparator change interrupts. The LED will be turned on by the interrupt handler when a rising edge on the comparator output is detected, and will be turned off when a falling edge is detected.

In order for this example to work properly, the ULED0 (JP22) jumper must be installed on the board.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 of Sandstorm-class Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata is fixed in Rev C2 of Sandstorm-class Stellaris microcontrollers.

GPIO (gpio_led)

This example application uses LEDs connected to GPIO pins to create a “roving eye” display. Port B0-B3 are driven in a sequential manner to give the illusion of an eye looking back and forth.

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), ULED2 (JP24), and ULED3 (JP25) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the LCD and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; individual LEDs connected to port B0-B2 will be turned on upon interrupt handler entry and off before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), and ULED2 (JP24) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 50 kHz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

DK-LM3S818 Quickstart Application (qs_dk-lm3s818)

This example uses the potentiometer on the development board to vary the rate of a repetitive beep from the piezo buzzer, while the light sensor will vary the frequency of the beep. Turning the knob in one direction will result in slower beeps while turning it in the other direction will result in faster beeps. The amount of light falling on the light sensor affects the frequency of the beep. The more light falling on the sensor the higher the pitch of the beep. The potentiometer setting along with the "note" representing the pitch of the beep is displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn the beeping noise on and off; when off the LCD and UART still provide the settings.

In the default jumper configuration of the development board, the push button will not actually mute the beep. In order for this example to fully work, the following jumper wire connections must be made: JP19 pin 2 to J6 pin 6.

SSI (ssi_atmel)

This example application uses the SSI master to communicate with the Atmel AT25F1024A EEPROM that is on the development board. The first 256 bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the SSI interrupt; since a 256-byte

read at a 1 MHz SSI bus speed takes around 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own GPIO (port B0 and B1) on each interrupt; the attached LED will indicate the occurrence and rate of interrupts.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (the SER0 connector on the Stellaris Family Development Board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port B0 is inverted so that it is easy to see that it is being fed, which occurs once every second.

39 DK-LM3S828 Example Applications

39.1 Introduction

The DK-LM3S828 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the Peripheral Device Controller on the Stellaris Family Development Kit board. The PDC is used to access the character LCD, eight user LEDs, eight user DIP switches, and twenty-four GPIOs.

There is an IAR workspace file (`dk-lm3s828.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`dk-lm3s828-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`dk-lm3s828.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/dk-lm3s828` subdirectory of the peripheral driver library source distribution.

39.2 API Functions

Functions

- unsigned char PDCDIPRead (void)
- unsigned char PDCGPIODirRead (unsigned char ucIdx)
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)
- unsigned char PDCGPIORead (unsigned char ucIdx)
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)
- void PDCInit (void)
- void PDCLCDBacklightOff (void)
- void PDCLCDBacklightOn (void)
- void PDCLCDClear (void)
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)
- void PDCLCDInit (void)
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)

- unsigned char PDCLEDRead (void)
- void PDCLEDWrite (unsigned char ucLED)
- unsigned char PDCRead (unsigned char ucAddr)
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)

39.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

39.2.2 Function Documentation

39.2.2.1 PDCDIPRead

Read the current value of the PDC DIP switches.

Prototype:

```
unsigned char
PDCDIPRead(void)
```

Description:

This function will read the current value of the DIP switches attached to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The current state of the DIP switches.

39.2.2.2 PDCGPIODirRead

Reads a GPIO direction register.

Prototype:

```
unsigned char
PDCGPIODirRead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO direction registers in the PDC. The direction bit is set for pins that are outputs and clear for pins that are inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the direction register.

39.2.2.3 PDCGPIODirWrite

Write a GPIO direction register.

Prototype:

```
void
PDCGPIODirWrite(unsigned char ucIdx,
                unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO direction register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO direction register.

Description:

This function writes ones of the GPIO direction registers in the PDC. The direction bit should be set for pins that are to be outputs and clear for pins that are to be inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

39.2.2.4 PDCGPIORead

Reads a GPIO data register.

Prototype:

```
unsigned char
PDCGPIORead(unsigned char ucIdx)
```

Parameters:

ucIdx is the index of the GPIO direction register to read; valid values are 0, 1, and 2.

Description:

This function reads one of the GPIO data registers in the PDC. The value returned for a pin is the value being driven out for outputs or the value being read for inputs.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The contents of the data register.

39.2.2.5 PDCGPIOWrite

Write a GPIO data register.

Prototype:

```
void
PDCGPIOWrite(unsigned char ucIdx,
             unsigned char ucValue)
```

Parameters:

ucIdx is the index of the GPIO data register to write; valid values are 0, 1, and 2.

ucValue is the value to write to the GPIO data register.

Description:

This function writes one of the GPIO direction registers in the PDC. The written to a pin is driven out for output pins and ignored for input pins.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

39.2.2.6 PDCInit

Initializes the connection to the PDC.

Prototype:

```
void
PDCInit(void)
```

Description:

This function will enable clocking to the SSI and GPIO A modules, configure the GPIO pins to be used for an SSI interface, and it will configure the SSI as a 1 Mbps master device, operating in MOTO mode. It will also enable the SSI module, and will enable the chip select for the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

39.2.2.7 PDCLCDBacklightOff

Turn off the backlight.

Prototype:

```
void
PDCLCDBacklightOff(void)
```

Description:

This function turns off the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

39.2.2.8 PDCLCDBacklightOn

Turns on the backlight.

Prototype:

```
void
PDCLCDBacklightOn(void)
```

Description:

This function turns on the backlight on the LCD.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

39.2.2.9 PDCLCDClear

Clear the screen.

Prototype:

```
void
PDCLCDClear(void)
```

Description:

This function clears the contents of the LCD screen. The cursor will be returned to the upper left corner.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

39.2.2.10 PDCLCDCreateChar

Write a character pattern to the LCD.

Prototype:

```
void
PDCLCDCreateChar(unsigned char ucChar,
                 unsigned char *pucData)
```

Parameters:

ucChar is the character index to create. Valid values are zero through seven.

pucData is the data for the character pattern. It contains eight bytes, with the first byte being the top row of the pattern. In each byte, the LSB is the right pixel of the pattern.

Description:

This function will write a character pattern into the LCD for use as a character to be displayed. After writing the pattern, it can be used on the LCD by writing the corresponding character index to the display.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

39.2.2.11 PDCLCDInit

Initializes the LCD display.

Prototype:
```
void
PDCLCDInit(void)
```

Description:
This function will set up the LCD display for writing. It will set the data bus to 8 bits, set the number of lines to 2, and the font size to 5x10. It will also turn the display off, clear the display, turn the display back on, and enable the backlight.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Note:
The PDC must be initialized via the PDCInit() function before this function can be called. Also, it may be necessary to adjust the contrast potentiometer in order to discern any output on the LCD display.

Returns:
None.

39.2.2.12 PDCLCDSetPos

Set the position of the cursor.

Prototype:
```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

Parameters:
ucX is the horizontal position. Valid values are zero through fifteen.
ucY is the vertical position.. Valid values are zero and one.

Description:
This function will move the cursor to the specified position. All characters written to the LCD are placed at the current cursor position, which is automatically advanced.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

39.2.2.13 PDCLCDWrite

Writes a string to the LCD display.

Prototype:

```
void
PDCLCDWrite(const char *pcStr,
            unsigned long ulCount)
```

Parameters:

pcStr pointer to the string to be displayed.

ulCount is the number of characters to be displayed.

Description:

This function will display a string on the LCD at the current cursor position. It is the caller's responsibility to position the cursor to the place where the string should be displayed (either explicitly via PDCLCDSetPos() or implicitly from where the cursor was left after a previous call to PDCLCDWrite()), and to properly account for the LCD boundary (line wrapping is not automatically performed). Null characters are not treated special and are written to the LCD, which interprets it as a special programmable character glyph (see PDCLCDCreateChar()).

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

39.2.2.14 PDCLEDRead

Read the current status of the PDC LEDs.

Prototype:

```
unsigned char
PDCLEDRead(void)
```

Description:

This function will read the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

The value currently displayed by the LEDs.

39.2.2.15 PDCLEDWrite

Write to the PDC LEDs.

Prototype:

```
void
PDCLEDWrite(unsigned char ucLED)
```

Parameters:

ucLED value to write to the LEDs.

Description:

This function set the state of the LEDs connected to the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

None.

39.2.2.16 PDCRead

Read a PDC register.

Prototype:

```
unsigned char
PDCRead(unsigned char ucAddr)
```

Parameters:

ucAddr specifies the PDC register to read.

Description:

This function will perform the SSI transfers required to read a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:

Returns the value read from the PDC.

39.2.2.17 PDCWrite

Write a PDC register.

Prototype:

```
void
PDCWrite(unsigned char ucAddr,
         unsigned char ucData)
```

Parameters:

ucAddr specifies the PDC register to write.

ucData specifies the data to write.

Description:

This function will perform the SSI transfers required to write a register in the PDC on the Stellaris development board.

This function is contained in `pdc.c`, with `pdc.h` containing the API definition for use by applications.

Returns:
None.

39.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 of Sandstorm-class Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata is fixed in Rev C2 of Sandstorm-class Stellaris microcontrollers.

GPIO (gpio_led)

This example application uses LEDs connected to GPIO pins to create a “roving eye” display. Port B0-B3 are driven in a sequential manner to give the illusion of an eye looking back and forth.

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), ULED2 (JP24), and ULED3 (JP25) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the LCD and is a starting point for more complicated applications.

I2C (i2c_atmel)

This example application uses the I2C master to communicate with the Atmel AT24C08A EEPROM that is on the development board. The first sixteen bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the I2C interrupt; since a sixteen-byte read at a 100 kHz I2C bus speed takes almost 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

In order for this example to work properly, the I2C_SCL (JP14), I2C_SDA (JP13), and I2CM_A2 (JP11) jumpers must be installed on the board, and the I2CM_WP (JP12) jumper must be removed.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; individual LEDs connected to port B0-B2 will be turned on upon interrupt handler entry and off before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

In order for this example to work properly, the ULED0 (JP22), ULED1 (JP23), and ULED2 (JP24) jumpers must be installed on the board, and the PB1 (JP1) jumper on the daughtercard must be set to pins 2 & 3.

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

DK-LM3S828 Quickstart Application (qs_dk-lm3s828)

This example uses the potentiometer on the development board to vary the rate of a click sound from the piezo buzzer. Turning the knob in one direction will result in slower clicks while turning it in the other direction will result in faster clicks. The potentiometer setting is displayed on the LCD, and a log of the readings is output on the UART at 115,200, 8-n-1. The push button can be used to turn the clicking noise on and off; when off the LCD and UART still provide the settings.

SSI (ssi_atmel)

This example application uses the SSI master to communicate with the Atmel AT25F1024A EEPROM that is on the development board. The first 256 bytes of the EEPROM are erased and then programmed with an incrementing sequence. The data is then read back to verify its correctness. The transfer is managed by an interrupt handler in response to the SSI interrupt; since a 256-byte read at a 1 MHz SSI bus speed takes around 2 ms, this allows a lot of other processing to occur during the transfer (though that time is not utilized by this example).

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own GPIO (port B0 and B1) on each interrupt; the attached LED will indicate the occurrence and rate of interrupts.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (the SER0 connector on the Stellaris Family Development Board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port B0 is inverted so that it is easy to see that it is being fed, which occurs once every second.

40 EK-LM3S1968 Example Applications

40.1 Introduction

The EK-LM3S1968 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the RiTdisplay 128x96 4-bit gray-scale OLED graphical display on the Stellaris LM3S1968 Evaluation Kit board.

There is also a board specific driver for the Class-D audio amplifier and speaker on the board. In order to use this driver, the system clock should be as high as possible and must be at least 256 KHz; increasing the system clock rate result in higher quality audio. This driver will play both 8-bit PCM data and 4-bit ADPCM data; the `converter` application (`converter.c` is the source code, and `converter.exe` is a pre-built binary) will take raw 16-bit signed PCM data and convert it into a C array that can be included in an application for playback purposes. For example, to encode `voice.pcm` with ADPCM and produce a C array called `g_pucVoice`:

```
converter -a -n g_pucVoice -o voice.h voice.pcm
```

To do the same, but encode to 8-bit PCM:

```
converter -p -n g_pucVoice -o voice.h voice.pcm
```

Since the Class-D audio driver will only play 8 KHz mono streams, and the `converter` application will only handle raw PCM input, an application such as `sox` will be needed to convert arbitrary wave files to the required format. To convert `voice.wav` to the required format for `converter`:

```
sox voice.wav -t raw -r 8000 -c 1 -s -w voice.pcm polyphase
```

The `polyphase` at the end selects a higher quality sample rate conversion algorithm. It may be helpful (and/or necessary) to also include `vol {factor}` before `polyphase` in order to increase the volume of the waveform. If `sox` complains of clipping, the volume needs to be reduced.

`sox` can be found at `http://sox.sourceforge.net`. There are numerous other audio applications (both open source and commercial) that can be used instead of `sox`.

There is an IAR workspace file (`ek-lm3s1968.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`ek-lm3s1968-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`ek-lm3s1968.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/ek-lm3s1968` subdirectory of the peripheral driver library source distribution.

40.2 API Functions

Functions

- tBoolean ClassDBusy (void)
- void ClassDInit (unsigned long ulPWMClock)
- void ClassDPlayADPCM (const unsigned char *pucBuffer, unsigned long ulLength)
- void ClassDPlayPCM (const unsigned char *pucBuffer, unsigned long ulLength)
- void ClassDPWMHandler (void)
- void ClassDStop (void)
- void ClassDVolumeDown (unsigned long ulVolume)
- void ClassDVolumeSet (unsigned long ulVolume)
- void ClassDVolumeUp (unsigned long ulVolume)
- void RIT128x96x4Clear (void)
- void RIT128x96x4Disable (void)
- void RIT128x96x4DisplayOff (void)
- void RIT128x96x4DisplayOn (void)
- void RIT128x96x4Enable (unsigned long ulFrequency)
- void RIT128x96x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)
- void RIT128x96x4Init (unsigned long ulFrequency)
- void RIT128x96x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY, unsigned char ucLevel)

40.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

40.2.2 Function Documentation

40.2.2.1 ClassDBusy

Determines if the Class-D audio driver is busy.

Prototype:

```
tBoolean
ClassDBusy(void)
```

Description:

This function determines if the Class-D audio driver is busy, either performing the startup or shutdown ramp for the speaker or playing an audio stream.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:
Returns **true** if the Class-D audio driver is busy and **false** otherwise.

40.2.2.2 ClassDInit

Initializes the Class-D audio driver.

Prototype:
```
void
ClassDInit(unsigned long ulPWMClock)
```

Parameters:
ulPWMClock is the rate of the clock supplied to the PWM module.

Description:
This function initializes the Class-D audio driver, preparing it to output audio data to the speaker.

The PWM module clock should be as high as possible; lower clock rates reduces the quality of the produced audio. For the best quality audio, the PWM module should be clocked at 50 MHz.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Note:
In order for the Class-D audio driver to function properly, the Class-D audio driver interrupt handler (ClassDPWMHandler()) must be installed into the vector table for the PWM1 interrupt.

Returns:
None.

40.2.2.3 ClassDPlayADPCM

Plays a buffer of 8 KHz IMA ADPCM data.

Prototype:
```
void
ClassDPlayADPCM(const unsigned char *pucBuffer,
                unsigned long ulLength)
```

Parameters:
pucBuffer is a pointer to the buffer containing the IMA ADPCM encoded data.
ulLength is the number of bytes in the buffer.

Description:
This function starts playback of a stream of IMA ADPCM encoded data. The data is decoded as needed and therefore does not require a large buffer in SRAM. This provides a 2:1 compression ratio relative to raw 8-bit PCM with little to no loss in audio quality.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:
None.

40.2.2.4 ClassDPlayPCM

Plays a buffer of 8 KHz, 8-bit, unsigned PCM data.

Prototype:
```
void
ClassDPlayPCM(const unsigned char *pucBuffer,
              unsigned long ulLength)
```

Parameters:
pucBuffer is a pointer to the buffer containing 8-bit, unsigned PCM data.

ulLength is the number of bytes in the buffer.

Description:
This function starts playback of a stream of 8-bit, unsigned PCM data. Since the data is unsigned, a value of 128 represents the mid-point of the speaker's travel (that is, corresponds to no DC offset).

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:
None.

40.2.2.5 ClassDPWMHandler

Handles the PWM1 interrupt.

Prototype:
```
void
ClassDPWMHandler(void)
```

Description:
This function responds to the PWM1 interrupt, updating the duty cycle of the output waveform in order to produce sound. It is the application's responsibility to ensure that this function is called in response to the PWM1 interrupt, typically by installing it in the vector table as the handler for the PWM1 interrupt.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:
None.

40.2.2.6 ClassDStop

Stops playback of the current audio stream.

Prototype:

```
void
ClassDStop(void)
```

Description:

This function immediately stops playback of the current audio stream. As a result, the output is changed directly to the mid-point, possibly resulting in a pop or click. It is then ramped down to no output, eliminating the current draw through the Class-D amplifier and speaker.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:

None.

40.2.2.7 ClassDVolumeDown

Decreases the volume of the audio playback.

Prototype:

```
void
ClassDVolumeDown(unsigned long ulVolume)
```

Parameters:

ulVolume is the amount by which to decrease the volume of the audio playback, specified as a value between 0 (for no adjustment) and 256 maximum adjustment).

Description:

This function decreases the volume of the audio playback relative to the current volume.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:

None.

40.2.2.8 ClassDVolumeSet

Sets the volume of the audio playback.

Prototype:

```
void
ClassDVolumeSet(unsigned long ulVolume)
```

Parameters:

ulVolume is the volume of the audio playback, specified as a value between 0 (for silence) and 256 (for full volume).

Description:

This function sets the volume of the audio playback. Setting the volume to 0 will mute the output, while setting the volume to 256 will play the audio stream without any volume adjustment (that is, full volume).

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:

None.

40.2.2.9 ClassDVolumeUp

Increases the volume of the audio playback.

Prototype:

```
void
ClassDVolumeUp(unsigned long ulVolume)
```

Parameters:

ulVolume is the amount by which to increase the volume of the audio playback, specified as a value between 0 (for no adjustment) and 256 maximum adjustment).

Description:

This function increases the volume of the audio playback relative to the current volume.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:

None.

40.2.2.10 RIT128x96x4Clear

Clears the OLED display.

Prototype:

```
void
RIT128x96x4Clear(void)
```

Description:

This function will clear the display RAM. All pixels in the display will be turned off.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

40.2.2.11 RIT128x96x4Disable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
RIT128x96x4Disable(void)
```

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

40.2.2.12 RIT128x96x4DisplayOff

Turns off the OLED display.

Prototype:

```
void
RIT128x96x4DisplayOff(void)
```

Description:

This function will turn off the OLED display. This will stop the scanning of the panel and turn off the on-chip DC-DC converter, preventing damage to the panel due to burn-in (it has similar characters to a CRT in this respect).

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

40.2.2.13 RIT128x96x4DisplayOn

Turns on the OLED display.

Prototype:

```
void
RIT128x96x4DisplayOn(void)
```

Description:

This function will turn on the OLED display, causing it to display the contents of its internal frame buffer.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

40.2.2.14 RIT128x96x4Enable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
RIT128x96x4Enable(unsigned long ulFrequency)
```

Parameters:

ulFrequency specifies the SSI Clock Frequency to be used.

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

40.2.2.15 RIT128x96x4ImageDraw

Displays an image on the OLED display.

Prototype:

```
void
RIT128x96x4ImageDraw(const unsigned char *pucImage,
                     unsigned long ulX,
                     unsigned long ulY,
                     unsigned long ulWidth,
                     unsigned long ulHeight)
```

Parameters:

pucImage is a pointer to the image data.

ulX is the horizontal position to display this image, specified in columns from the left edge of the display.

ulY is the vertical position to display this image, specified in rows from the top of the display.

ulWidth is the width of the image, specified in columns.

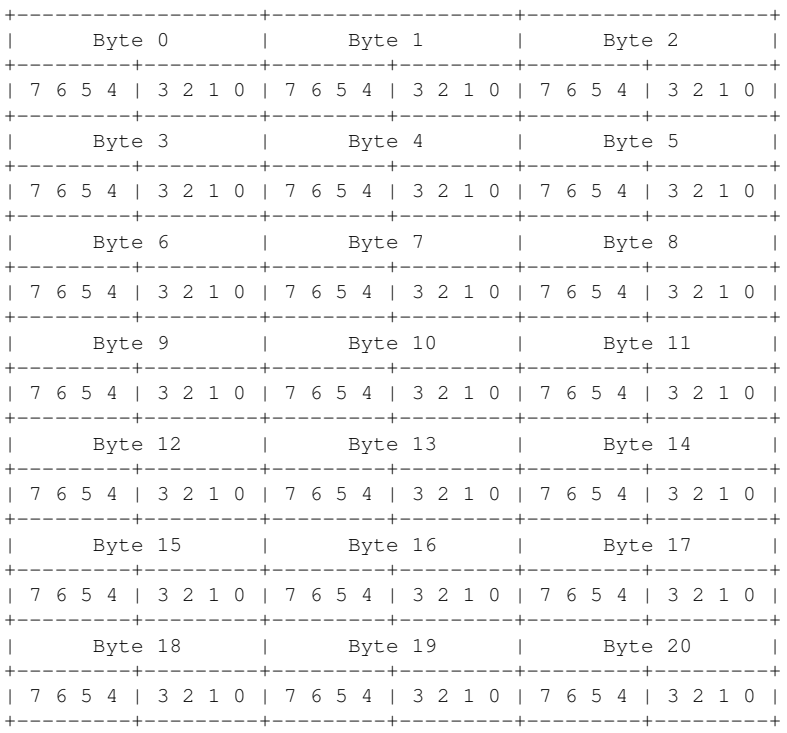
ulHeight is the height of the image, specified in rows.

Description:

This function will display a bitmap graphic on the display. Because of the format of the display RAM, the starting column (*ulX*) and the number of columns (*ulWidth*) must be an integer multiple of two.

The image data is organized with the first row of image data appearing left to right, followed immediately by the second row of image data. Each byte contains the data for two columns in the current row, with the leftmost column being contained in bits 7:4 and the rightmost column being contained in bits 3:0.

For example, an image six columns wide and seven scan lines tall would be arranged as follows (showing how the twenty one bytes of the image would appear on the display):



This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:
None.

40.2.2.16 RIT128x96x4Init

Initialize the OLED display.

Prototype:
```
void
RIT128x96x4Init(unsigned long ulFrequency)
```

Parameters:
ulFrequency specifies the SSI Clock Frequency to be used.

Description:
This function initializes the SSI interface to the OLED display and configures the SSD1329 controller on the panel.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:
None.

40.2.2.17 RIT128x96x4StringDraw

Displays a string on the OLED display.

Prototype:

```
void
RIT128x96x4StringDraw(const char *pcStr,
                      unsigned long ulX,
                      unsigned long ulY,
                      unsigned char ucLevel)
```

Parameters:

pcStr is a pointer to the string to display.

ulX is the horizontal position to display the string, specified in columns from the left edge of the display.

ulY is the vertical position to display the string, specified in rows from the top edge of the display.

ucLevel is the 4-bit gray scale value to be used for displayed text.

Description:

This function will draw a string on the display. Only the ASCII characters between 32 (space) and 126 (tilde) are supported; other characters will result in random data being draw on the display (based on whatever appears before/after the font in memory). The font is mono-spaced, so characters such as “i” and “l” have more white space around them than characters such as “m” or “w”.

If the drawing of the string reaches the right edge of the display, no more characters will be drawn. Therefore, special care is not required to avoid supplying a string that is “too long” to display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Note:

Because the OLED display packs 2 pixels of data in a single byte, the parameter *ulX* must be an even column number (for example, 0, 2, 4, and so on).

Returns:

None.

40.3 Examples

Audio Playback (audio)

This example application plays audio via the Class-D amplifier and speaker. The same source audio clip is provided in both PCM and ADPCM format so that the audio quality can be compared.

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the select push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO.

Graphics Example (graphics)

A simple application that displays scrolling text on the top line of the OLED display, along with a 4-bit gray scale image.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the OLED and is a starting point for more complicated applications.

Hibernate Example (hibernate)

An example to demonstrate the use of the Hibernation module. The user can put the microcontroller in hibernation by pressing the select button. The microcontroller will then wake on its own after 5 seconds, or immediately if the user presses the select button again. The program keeps a count of the number of times it has entered hibernation. The value of the counter is stored in the battery backed memory of the Hibernation module so that it can be retrieved when the microcontroller wakes.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the OLED; GPIO pins B0, B1 and B2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 440 Hz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

EK-LM3S1968 Quickstart Application (qs_ek-lm3s1968)

A game in which a blob-like character tries to find its way out of a maze. The character starts in the middle of the maze and must find the exit, which will always be located at one of the four corners of the maze. Once the exit to the maze is located, the character is placed into the middle of a new maze and must find the exit to that maze; this repeats endlessly.

The game is started by pressing the select push button on the right side of the board. During game play, the select push button will fire a bullet in the direction the character is currently facing, and the navigation push buttons on the left side of the board will cause the character to walk in the corresponding direction.

Populating the maze are a hundred spinning stars that mindlessly attack the character. Contact with one of these stars results in the game ending, but the stars go away when shot.

Score is accumulated for shooting the stars and for finding the exit to the maze. The game lasts for only one character, and the score is displayed on the virtual UART at 115,200, 8-N-1 during game play and will be displayed on the screen at the end of the game.

Since the OLED display on the evaluation board has burn-in characteristics similar to a CRT, the application also contains a screen saver. The screen saver will only become active if two minutes have passed without the user push button being pressed while waiting to start the game (that is, it will never come on during game play). Qix-style bouncing lines are drawn on the display by the screen saver.

After two minutes of running the screen saver, the processor will enter hibernation mode, and the red LED will turn on. Hibernation mode will be exited by pressing the select push button. The select push button will then need to be pressed again to start the game.

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own indicator on the display.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (connected to the FTDI virtual serial port on the evaluation board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED is inverted so that it is easy to see that it is being fed, which occurs once every second.

41 EK-LM3S2965 Example Applications

41.1 Introduction

The EK-LM3S2965 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the OSRAM 128x64 4-bit gray-scale OLED graphical display on the Stellaris LM3S2965 Evaluation Kit boards.

These examples and display driver are for the EK-LM3S2965 Rev A boards, which utilize the 128x64 OSRAM display. The Rev A boards can be identified by looking on the back of the circuit board opposite the JTAG header. The board part number is located there and will end with an "A". If the board part number ends with a "C", then refer instead to the examples chapter for the EK-LM3S2965 Rev C Example Applications.

There is an IAR workspace file (`ek-lm3s2965.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`ek-lm3s2965-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`ek-lm3s2965.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/ek-lm3s2965` subdirectory of the peripheral driver library source distribution.

41.2 API Functions

Functions

- void OSRAM128x64x4Clear (void)
- void OSRAM128x64x4Disable (void)
- void OSRAM128x64x4DisplayOff (void)
- void OSRAM128x64x4DisplayOn (void)
- void OSRAM128x64x4Enable (unsigned long ulFrequency)
- void OSRAM128x64x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)
- void OSRAM128x64x4Init (unsigned long ulFrequency)

- void OSRAM128x64x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY, unsigned char ucLevel)

41.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

41.2.2 Function Documentation

41.2.2.1 OSRAM128x64x4Clear

Clears the OLED display.

Prototype:

```
void
OSRAM128x64x4Clear(void)
```

Description:

This function will clear the display RAM. All pixels in the display will be turned off.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:

None.

41.2.2.2 OSRAM128x64x4Disable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
OSRAM128x64x4Disable(void)
```

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:

None.

41.2.2.3 OSRAM128x64x4DisplayOff

Turns off the OLED display.

Prototype:

```
void
OSRAM128x64x4DisplayOff(void)
```

Description:

This function will turn off the OLED display. This will stop the scanning of the panel and turn off the on-chip DC-DC converter, preventing damage to the panel due to burn-in (it has similar characters to a CRT in this respect).

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:

None.

41.2.2.4 OSRAM128x64x4DisplayOn

Turns on the OLED display.

Prototype:

```
void
OSRAM128x64x4DisplayOn(void)
```

Description:

This function will turn on the OLED display, causing it to display the contents of its internal frame buffer.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:

None.

41.2.2.5 OSRAM128x64x4Enable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
OSRAM128x64x4Enable(unsigned long ulFrequency)
```

Parameters:

ulFrequency specifies the SSI Clock Frequency to be used.

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:

None.

41.2.2.6 OSRAM128x64x4ImageDraw

Displays an image on the OLED display.

Prototype:

```
void
OSRAM128x64x4ImageDraw(const unsigned char *pucImage,
                       unsigned long ulX,
                       unsigned long ulY,
                       unsigned long ulWidth,
                       unsigned long ulHeight)
```

Parameters:

pucImage is a pointer to the image data.

ulX is the horizontal position to display this image, specified in columns from the left edge of the display.

ulY is the vertical position to display this image, specified in rows from the top of the display.

ulWidth is the width of the image, specified in columns.

ulHeight is the height of the image, specified in rows.

Description:

This function will display a bitmap graphic on the display. Because of the format of the display RAM, the starting column (*ulX*) and the number of columns (*ulWidth*) must be an integer multiple of two.

The image data is organized with the first row of image data appearing left to right, followed immediately by the second row of image data. Each byte contains the data for two columns in the current row, with the leftmost column being contained in bits 7:4 and the rightmost column being contained in bits 3:0.

For example, an image six columns wide and seven scan lines tall would be arranged as follows (showing how the twenty one bytes of the image would appear on the display):

```
+-------------------+-------------------+-------------------+
|      Byte 0       |      Byte 1       |      Byte 2       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 3       |      Byte 4       |      Byte 5       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 6       |      Byte 7       |      Byte 8       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 9       |      Byte 10      |      Byte 11      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 12      |      Byte 13      |      Byte 14      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 15      |      Byte 16      |      Byte 17      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 18      |      Byte 19      |      Byte 20      |
```

```
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
```

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:
None.

41.2.2.7 OSRAM128x64x4Init

Initialize the OLED display.

Prototype:
```
void
OSRAM128x64x4Init(unsigned long ulFrequency)
```

Parameters:
ulFrequency specifies the SSI Clock Frequency to be used.

Description:
This function initializes the SSI interface to the OLED display and configures the SSD0323 controller on the panel.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:
None.

41.2.2.8 OSRAM128x64x4StringDraw

Displays a string on the OLED display.

Prototype:
```
void
OSRAM128x64x4StringDraw(const char *pcStr,
                        unsigned long ulX,
                        unsigned long ulY,
                        unsigned char ucLevel)
```

Parameters:
pcStr is a pointer to the string to display.
ulX is the horizontal position to display the string, specified in columns from the left edge of the display.
ulY is the vertical position to display the string, specified in rows from the top edge of the display.
ucLevel is the 4-bit gray scale value to be used for displayed text.

Description:

This function will draw a string on the display. Only the ASCII characters between 32 (space) and 126 (tilde) are supported; other characters will result in random data being draw on the display (based on whatever appears before/after the font in memory). The font is mono-spaced, so characters such as “i” and “l” have more white space around them than characters such as “m” or “w”.

If the drawing of the string reaches the right edge of the display, no more characters will be drawn. Therefore, special care is not required to avoid supplying a string that is “too long” to display.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Note:

Because the OLED display packs 2 pixels of data in a single byte, the parameter *ulX* must be an even column number (for example, 0, 2, 4, and so on).

Returns:

None.

41.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

CAN Device Board LED Application (can_device_led)

This simple application uses the two buttons on the board as a light switch. When the "up" button is pressed the status LED will turn on. When the "down" button is pressed, the status LED will turn off.

CAN Device Board Quickstart Application (can_device_qs)

This application uses the CAN controller to communicate with the evaluation board that is running the example game. It receives messages over CAN to turn on, turn off, or to pulse the LED on the device board. It also sends CAN messages when either of the up and down buttons are pressed or released.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the select push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO.

Graphics Example (graphics)

A simple application that displays scrolling text on the top line of the OLED display, along with a 4-bit gray scale image.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the OLED and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the OLED; GPIO pins B0, B1 and B2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 440 Hz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

EK-LM3S2965 Quickstart Application (qs_ek-lm3s2965)

A game in which a blob-like character tries to find its way out of a maze. The character starts in the middle of the maze and must find the exit, which will always be located at one of the four corners of the maze. Once the exit to the maze is located, the character is placed into the middle of a new maze and must find the exit to that maze; this repeats endlessly.

The game is started by pressing the select push button on the right side of the board. During game play, the select push button will fire a bullet in the direction the character is currently facing, and the navigation push buttons on the left side of the board will cause the character to walk in the corresponding direction.

Populating the maze are a hundred spinning stars that mindlessly attack the character. Contact with one of these stars results in the game ending, but the stars go away when shot.

Score is accumulated for shooting the stars and for finding the exit to the maze. The game lasts for only one character, and the score is displayed on the virtual UART at 115,200, 8-N-1 during game play and will be displayed on the screen at the end of the game.

If the CAN device board is attached and is running the can_device_qs application, the volume of the music and sound effects can be adjusted over CAN with the two push buttons on the target board. The LED on the CAN device board will track the state of the LED on the main board via CAN messages. The operation of the game will not be affected by the absence of the CAN device board.

Since the OLED display on the evaluation board has burn-in characteristics similar to a CRT, the application also contains a screen saver. The screen saver will only become active if two minutes have passed without the user push button being pressed while waiting to start the game (that is, it will never come on during game play). Qix-style bouncing lines are drawn on the display by the screen saver.

After two minutes of running the screen saver, the display will be turned off and the user LED will blink. Either mode of screen saver (bouncing lines or blank display) will be exited by pressing the select push button. The select push button will then need to be pressed again to start the game.

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own indicator on the display.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (connected to the FTDI virtual serial port on the evaluation board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED is inverted so that it is easy to see that it is being fed, which occurs once every second.

42 EK-LM3S2965 Rev C Example Applications

42.1 Introduction

The EK-LM3S2965 Rev C example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the RiTdisplay 128x96 4-bit gray-scale OLED graphical display on the Stellaris LM3S2965 Rev C Evaluation Kit boards.

These examples and display driver are for the EK-LM3S2965 Rev C boards, which utilize the 128x96 RiTdisplay display. The Rev C boards can be identified by looking on the back of the circuit board opposite the JTAG header. The board part number is located there and will end with an "C". If the board part number ends with a "A", then refer instead to the examples chapter for the EK-LM3S2965 Example Applications.

There is an IAR workspace file (`ek-lm3s2965_revc.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`ek-lm3s2965_revc-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`ek-lm3s2965_revc.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/ek-lm3s2965_revc` subdirectory of the peripheral driver library source distribution.

42.2 API Functions

Functions

- void RIT128x96x4Clear (void)
- void RIT128x96x4Disable (void)
- void RIT128x96x4DisplayOff (void)
- void RIT128x96x4DisplayOn (void)
- void RIT128x96x4Enable (unsigned long ulFrequency)
- void RIT128x96x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)
- void RIT128x96x4Init (unsigned long ulFrequency)

- void RIT128x96x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY, unsigned char ucLevel)

42.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

42.2.2 Function Documentation

42.2.2.1 RIT128x96x4Clear

Clears the OLED display.

Prototype:

```
void
RIT128x96x4Clear(void)
```

Description:

This function will clear the display RAM. All pixels in the display will be turned off.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

42.2.2.2 RIT128x96x4Disable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
RIT128x96x4Disable(void)
```

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

42.2.2.3 RIT128x96x4DisplayOff

Turns off the OLED display.

Prototype:

```
void
RIT128x96x4DisplayOff(void)
```

Description:

This function will turn off the OLED display. This will stop the scanning of the panel and turn off the on-chip DC-DC converter, preventing damage to the panel due to burn-in (it has similar characters to a CRT in this respect).

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

42.2.2.4 RIT128x96x4DisplayOn

Turns on the OLED display.

Prototype:

```
void
RIT128x96x4DisplayOn(void)
```

Description:

This function will turn on the OLED display, causing it to display the contents of its internal frame buffer.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

42.2.2.5 RIT128x96x4Enable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
RIT128x96x4Enable(unsigned long ulFrequency)
```

Parameters:

ulFrequency specifies the SSI Clock Frequency to be used.

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

42.2.2.6 RIT128x96x4ImageDraw

Displays an image on the OLED display.

Prototype:

```
void
RIT128x96x4ImageDraw(const unsigned char *pucImage,
                     unsigned long ulX,
                     unsigned long ulY,
                     unsigned long ulWidth,
                     unsigned long ulHeight)
```

Parameters:

pucImage is a pointer to the image data.

ulX is the horizontal position to display this image, specified in columns from the left edge of the display.

ulY is the vertical position to display this image, specified in rows from the top of the display.

ulWidth is the width of the image, specified in columns.

ulHeight is the height of the image, specified in rows.

Description:

This function will display a bitmap graphic on the display. Because of the format of the display RAM, the starting column (*ulX*) and the number of columns (*ulWidth*) must be an integer multiple of two.

The image data is organized with the first row of image data appearing left to right, followed immediately by the second row of image data. Each byte contains the data for two columns in the current row, with the leftmost column being contained in bits 7:4 and the rightmost column being contained in bits 3:0.

For example, an image six columns wide and seven scan lines tall would be arranged as follows (showing how the twenty one bytes of the image would appear on the display):

```
+-------------------+-------------------+-------------------+
|      Byte 0       |      Byte 1       |      Byte 2       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 3       |      Byte 4       |      Byte 5       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 6       |      Byte 7       |      Byte 8       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 9       |      Byte 10      |      Byte 11      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 12      |      Byte 13      |      Byte 14      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 15      |      Byte 16      |      Byte 17      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 18      |      Byte 19      |      Byte 20      |
```

```
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
```

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:
None.

42.2.2.7 RIT128x96x4Init

Initialize the OLED display.

Prototype:

```
void
RIT128x96x4Init(unsigned long ulFrequency)
```

Parameters:
ulFrequency specifies the SSI Clock Frequency to be used.

Description:
This function initializes the SSI interface to the OLED display and configures the SSD1329 controller on the panel.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:
None.

42.2.2.8 RIT128x96x4StringDraw

Displays a string on the OLED display.

Prototype:

```
void
RIT128x96x4StringDraw(const char *pcStr,
                      unsigned long ulX,
                      unsigned long ulY,
                      unsigned char ucLevel)
```

Parameters:
pcStr is a pointer to the string to display.
ulX is the horizontal position to display the string, specified in columns from the left edge of the display.
ulY is the vertical position to display the string, specified in rows from the top edge of the display.
ucLevel is the 4-bit gray scale value to be used for displayed text.

Description:

This function will draw a string on the display. Only the ASCII characters between 32 (space) and 126 (tilde) are supported; other characters will result in random data being draw on the display (based on whatever appears before/after the font in memory). The font is mono-spaced, so characters such as “i” and “l” have more white space around them than characters such as “m” or “w”.

If the drawing of the string reaches the right edge of the display, no more characters will be drawn. Therefore, special care is not required to avoid supplying a string that is “too long” to display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Note:

Because the OLED display packs 2 pixels of data in a single byte, the parameter *ulX* must be an even column number (for example, 0, 2, 4, and so on).

Returns:

None.

42.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

CAN Device Board LED Application (can_device_led)

This simple application uses the two buttons on the board as a light switch. When the "up" button is pressed the status LED will turn on. When the "down" button is pressed, the status LED will turn off.

CAN Device Board Quickstart Application (can_device_qs)

This application uses the CAN controller to communicate with the evaluation board that is running the example game. It receives messages over CAN to turn on, turn off, or to pulse the LED on the device board. It also sends CAN messages when either of the up and down buttons are pressed or released.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the select push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO.

Graphics Example (graphics)

A simple application that displays scrolling text on the top line of the OLED display, along with a 4-bit gray scale image.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the OLED and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the OLED; GPIO pins B0, B1 and B2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 440 Hz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

EK-LM3S2965 Rev C Quickstart Application (qs_ek-lm3s2965_revc)

A game in which a blob-like character tries to find its way out of a maze. The character starts in the middle of the maze and must find the exit, which will always be located at one of the four corners of the maze. Once the exit to the maze is located, the character is placed into the middle of a new maze and must find the exit to that maze; this repeats endlessly.

The game is started by pressing the select push button on the right side of the board. During game play, the select push button will fire a bullet in the direction the character is currently facing, and the navigation push buttons on the left side of the board will cause the character to walk in the corresponding direction.

Populating the maze are a hundred spinning stars that mindlessly attack the character. Contact with one of these stars results in the game ending, but the stars go away when shot.

Score is accumulated for shooting the stars and for finding the exit to the maze. The game lasts for only one character, and the score is displayed on the virtual UART at 115,200, 8-N-1 during game play and will be displayed on the screen at the end of the game.

If the CAN device board is attached and is running the can_device_qs application, the volume of the music and sound effects can be adjusted over CAN with the two push buttons on the target board. The LED on the CAN device board will track the state of the LED on the main board via CAN messages. The operation of the game will not be affected by the absence of the CAN device board.

Since the OLED display on the evaluation board has burn-in characteristics similar to a CRT, the application also contains a screen saver. The screen saver will only become active if two minutes have passed without the user push button being pressed while waiting to start the game (that is, it will never come on during game play). Qix-style bouncing lines are drawn on the display by the screen saver.

After two minutes of running the screen saver, the display will be turned off and the user LED will blink. Either mode of screen saver (bouncing lines or blank display) will be exited by pressing the select push button. The select push button will then need to be pressed again to start the game.

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own indicator on the display.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (connected to the FTDI virtual serial port on the evaluation board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED is inverted so that it is easy to see that it is being fed, which occurs once every second.

43 EK-LM3S3748 Example Applications

43.1 Introduction

The EK-LM3S3748 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There are board specific drivers for the Formike Electronic 128x128 color CSTN graphical display and Class-D audio amplifier and speaker on the Stellaris EK-LM3S3748 Evaluation Kit boards.

There is an IAR workspace file (`ek-lm3s3748.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`ek-lm3s3748-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`ek-lm3s3748.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/ek-lm3s3748` subdirectory of the peripheral driver library source distribution.

43.2 API Functions

Functions

- void ButtonsInit (void)
- unsigned char ButtonsPoll (unsigned char *pucDelta, unsigned char *pucRepeat)
- void ButtonsSetAutoRepeat (unsigned char ucButtonIDs, unsigned char ucInitialTicks, unsigned char ucRepeatTicks)
- tBoolean ClassDBusy (void)
- void ClassDInit (unsigned long ulPWMClock)
- void ClassDPlayADPCM (const unsigned char *pucBuffer, unsigned long ulLength)
- void ClassDPlayPCM (const unsigned char *pucBuffer, unsigned long ulLength)
- void ClassDPWMHandler (void)
- void ClassDStop (void)
- void ClassDVolumeDown (unsigned long ulVolume)
- void ClassDVolumeSet (unsigned long ulVolume)
- void ClassDVolumeUp (unsigned long ulVolume)
- void Formike128x128x16BacklightOff (void)

- void Formike128x128x16BacklightOn (void)
- void Formike128x128x16Init (void)

Variables

- const tDisplay g_sFormike128x128x16

43.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

43.2.2 Function Documentation

43.2.2.1 ButtonsInit

Initializes the GPIO pins used by the board pushbuttons.

Prototype:

```
void
ButtonsInit(void)
```

Description:

This function must be called during application initialization to configure the GPIO pins to which the pushbuttons are attached. It enables the port used by the buttons and configures each button GPIO as an input with a weak pull-up.

This function is contained in `buttons.c`, with `buttons.h` containing the API definition for use by applications.

Returns:

None.

43.2.2.2 ButtonsPoll

Polls the current state of the buttons and determines which have changed.

Prototype:

```
unsigned char
ButtonsPoll(unsigned char *pucDelta,
            unsigned char *pucRepeat)
```

Parameters:

pucDelta points to a character that will be written to indicate which button states changed since the last time this function was called. This value is derived from the debounced state of the buttons.

pucRepeat points to a character that will be written to indicate which buttons are signalling an autorepeat as a result of this call.

Description:

This function should be called periodically by the application to poll the pushbuttons. It determines both which buttons have changed state since the last call and also signals autorepeats based on the button state and the number of ButtonsPoll() calls made since the button was last pressed.

Autorepeats are signalled at an application-specified rate if a key has been held down for longer than an initial delay period. To ensure that autorepeats are generated at the desired rate, the application should ensure that this function is called at a regular period since all autorepeat timing is calculated in terms of calls to ButtonsPoll().

This function is contained in `buttons.c`, with `buttons.h` containing the API definition for use by applications.

Returns:

Returns the current debounced state of the buttons where a 1 in the button ID's position indicates that the button is released and a 0 indicates that it is pressed.

43.2.2.3 ButtonsSetAutoRepeat

Sets the autorepeat parameters for one or more buttons.

Prototype:

```
void
ButtonsSetAutoRepeat(unsigned char ucButtonIDs,
                     unsigned char ucInitialTicks,
                     unsigned char ucRepeatTicks)
```

Parameters:

ucButtonIDs is a bitmask containing the ORed IDs of the buttons whose autorepeat parameters are to be set.

ucInitialTicks is the number of ticks (calls to ButtonsPoll()) before the first autorepeat is reported for the key if it is pressed for an extended period.

ucRepeatTicks is the number of ticks that must elapse after the initial period (*ucInitialTicks*) has expired between each subsequent autorepeat is reported for the key.

Description:

This function may be called to change the autorepeat delay and repeat period for one or more keys. Autorepeat allows an application to be signalled periodically if any key is held down for an extended period of time. After an initial delay following the original button press, a repeat signal flag is generated at a period determined by *ucRepeatTicks* and the interval between calls to ButtonsPoll().

For example, to configure a button such that it starts autorepeating 500mS after it is initially pressed and signals an autorepeat every 100mS until it is released, and assuming that ButtonsPoll() is called every 50mS, the following parameters would be used:

```
ucInitialTicks = 10
```

```
ucRepeatTicks = 2
```

This function is contained in `buttons.c`, with `buttons.h` containing the API definition for use by applications.

Returns:
None.

43.2.2.4 ClassDBusy

Determines if the Class-D audio driver is busy.

Prototype:
```
tBoolean
ClassDBusy(void)
```

Description:
This function determines if the Class-D audio driver is busy, either performing the startup or shutdown ramp for the speaker or playing an audio stream.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:
Returns **true** if the Class-D audio driver is busy and **false** otherwise.

43.2.2.5 ClassDInit

Initializes the Class-D audio driver.

Prototype:
```
void
ClassDInit(unsigned long ulPWMClock)
```

Parameters:
ulPWMClock is the rate of the clock supplied to the PWM module.

Description:
This function initializes the Class-D audio driver, preparing it to output audio data to the speaker.

The PWM module clock should be as high as possible; lower clock rates reduces the quality of the produced audio. For the best quality audio, the PWM module should be clocked at 50 MHz.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Note:
In order for the Class-D audio driver to function properly, the Class-D audio driver interrupt handler (ClassDPWMHandler()) must be installed into the vector table for the PWM1 interrupt.

Returns:
None.

43.2.2.6 ClassDPlayADPCM

Plays a buffer of 8 KHz IMA ADPCM data.

Prototype:

```
void
ClassDPlayADPCM(const unsigned char *pucBuffer,
                unsigned long ulLength)
```

Parameters:

pucBuffer is a pointer to the buffer containing the IMA ADPCM encoded data.

ulLength is the number of bytes in the buffer.

Description:

This function starts playback of a stream of IMA ADPCM encoded data. The data is decoded as needed and therefore does not require a large buffer in SRAM. This provides a 2:1 compression ratio relative to raw 8-bit PCM with little to no loss in audio quality.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:

None.

43.2.2.7 ClassDPlayPCM

Plays a buffer of 8 KHz, 8-bit, unsigned PCM data.

Prototype:

```
void
ClassDPlayPCM(const unsigned char *pucBuffer,
              unsigned long ulLength)
```

Parameters:

pucBuffer is a pointer to the buffer containing 8-bit, unsigned PCM data.

ulLength is the number of bytes in the buffer.

Description:

This function starts playback of a stream of 8-bit, unsigned PCM data. Since the data is unsigned, a value of 128 represents the mid-point of the speaker's travel (that is, corresponds to no DC offset).

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:

None.

43.2.2.8 ClassDPWMHandler

Handles the PWM1 interrupt.

Prototype:

```
void
ClassDPWMHandler(void)
```

Description:

This function responds to the PWM1 interrupt, updating the duty cycle of the output waveform in order to produce sound. It is the application's responsibility to ensure that this function is called in response to the PWM1 interrupt, typically by installing it in the vector table as the handler for the PWM1 interrupt.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:

None.

43.2.2.9 ClassDStop

Stops playback of the current audio stream.

Prototype:

```
void
ClassDStop(void)
```

Description:

This function immediately stops playback of the current audio stream. As a result, the output is changed directly to the mid-point, possibly resulting in a pop or click. It is then ramped down to no output, eliminating the current draw through the Class-D amplifier and speaker.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:

None.

43.2.2.10 ClassDVolumeDown

Decreases the volume of the audio playback.

Prototype:

```
void
ClassDVolumeDown(unsigned long ulVolume)
```

Parameters:

ulVolume is the amount by which to decrease the volume of the audio playback, specified as a value between 0 (for no adjustment) and 256 maximum adjustment).

Description:

This function decreases the volume of the audio playback relative to the current volume.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:
None.

43.2.2.11 ClassDVolumeSet

Sets the volume of the audio playback.

Prototype:
```
void
ClassDVolumeSet(unsigned long ulVolume)
```

Parameters:
ulVolume is the volume of the audio playback, specified as a value between 0 (for silence) and 256 (for full volume).

Description:
This function sets the volume of the audio playback. Setting the volume to 0 will mute the output, while setting the volume to 256 will play the audio stream without any volume adjustment (that is, full volume).

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:
None.

43.2.2.12 ClassDVolumeUp

Increases the volume of the audio playback.

Prototype:
```
void
ClassDVolumeUp(unsigned long ulVolume)
```

Parameters:
ulVolume is the amount by which to increase the volume of the audio playback, specified as a value between 0 (for no adjustment) and 256 maximum adjustment).

Description:
This function increases the volume of the audio playback relative to the current volume.

This function is contained in `class-d.c`, with `class-d.h` containing the API definition for use by applications.

Returns:
None.

43.2.2.13 Formike128x128x16BacklightOff

Turns off the backlight.

Prototype:
```
void
Formike128x128x16BacklightOff(void)
```

Description:
This function turns off the backlight on the display.

Returns:
None.

43.2.2.14 Formike128x128x16BacklightOn

Turns on the backlight.

Prototype:
```
void
Formike128x128x16BacklightOn(void)
```

Description:
This function turns on the backlight on the display.

Returns:
None.

43.2.2.15 Formike128x128x16Init

Initializes the display driver.

Prototype:
```
void
Formike128x128x16Init(void)
```

Description:
This function initializes the ST7637 display controller on the panel, preparing it to display data.

Returns:
None.

43.2.3 Variable Documentation

43.2.3.1 g_sFormike128x128x16

Definition:
```
const tDisplay g_sFormike128x128x16
```

Description:
The display structure that describes the driver for the Formike Electronic KWH015C04-F01 CSTN panel with an ST7637 controller.

43.3 Examples

Audio Playback (audio)

This example application plays audio via the Class-D amplifier and speaker. The same source audio clip is provided in both PCM and ADPCM format so that the audio quality can be compared.

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the ROM-based boot loader. At startup, the application will configure the UART and branch to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the ROM-based boot loader. At startup, the application will configure the UART, wait for the select button to be pressed, and then branch to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the select push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the

push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, four pins (PC0, PC1, PC2, and PC3) are switched.

Graphics Library Demonstration (grlib_demo)

This application provides a demonstration of the capabilities of the Stellaris Graphics Library. The display will be configured to demonstrate the available drawing primitives: lines, circles, rectangles, strings, and images.

Hello World (hello)

A very simple “hello world” example. It simply displays “Hello World!” on the display and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the display; GPIO pins D0, D1 and D2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 20% duty cycle PWM signal and a 80% duty cycle PWM signal, both at 8000 Hz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

Quickstart Oscilloscope (qs-scope)

A two channel oscilloscope implemented using the Stellaris microcontroller’s analog-to-digital converter (ADC). The oscilloscope supports sample rates of up to 1M sample per second and will show the captured waveforms on the CSTN display. On-screen menus provide user control over timebase, channel voltage scale and position, trigger type, trigger level and trigger position. Other

features include the ability to save captured data as comma-separated-value files suitable for use with spreadsheet applications or bitmap images on either an installed microSD card or a USB flash drive. The board may also be connected to a WindowsXP or Vista host system and controlled remotely using a Windows application.

Oscilloscope User Interface

All oscilloscope controls and settings are accessed using the navigation control on the board. This control offers up, down, left, right and select functions in a single unit. Rocking the control in the desired direction sends "up", "down", "left" or "right" messages to the application and pressing on the center sends a "select" message.

Controls and settings are arranged into groups by function such as display settings, trigger settings, file operations and setup choices. These groups are accessed by pressing "select" to display the main menu. With the menu displayed, use "up" and "down" to select between the available groups. When the desired group is highlighted, press "select" once again to dismiss the menu.

Controls from the currently selected group are shown in the bottom portion of the application display. Use "up" and "down" to cycle through the controls in the group and "left" and "right" to change the value of, or select the action associated with the control which is currently displayed.

The control groups and the individual controls offered by each are outlined below:

```
Group    Control       Setting
-----    -------       -------

Display
         Channel 2     ON or OFF.
         Timebase      Select values from 2uS to 50mS per division.
         Ch1 Scale     Select values from 100mV to 10V per division.
         Ch2 Scale     Select values from 100mV to 10V per division.
         Ch1 Offset    Press and hold "left" or "right" to move the
                       waveform up or down in 100mV increments.
         Ch2 Offset    Press and hold "left" or "right" to move the
                       waveform up or down in 100mV increments.
Trigger
         Trigger       The trigger type - Always, Rising, Falling or
                       Level.
         Trig Channel  1 or 2 to select the channel used for triggering.
         Trig Level    Press and hold "left" or "right" to change the
                       trigger level in 100mV increments.
         Trig Pos      Press and hold "left" or "right" to move the
                       trigger position on the display.
         Mode          Running or Stopped.
         One Shot      If the current mode is "Stopped", pressing
                       "left" or "right" initiates capture and display
                       of a single waveform.
Setup
         Captions      Select ON to show the timebase and scale
                       captions or OFF to remove them from the display.
         Voltages      Select ON to show the measured voltages for each
                       channel or OFF to remove them from the display.
         Grid          Select ON to show the graticule lines or OFF to
                       remove them from the display.
         Ground        Select ON to show dotted lines corresponding
                       to the ground levels for each channel or OFF
                       to remove them from the display.
         Trig Level    Select ON to show a solid horizontal line
                       corresponding to the trigger level for the
                       trigger channel or OFF to remove this line from
                       the display.
         Trig Pos      Select ON to show a solid vertical line at the
                       trigger position or OFF to remove this line from
```

```
                    the display.
          Clicks    Select ON to enable sounds on button presses or
                    OFF to disable them.
          USB Mode  Select Host to operate in USB host mode and
                    allow use of flash memory sticks, or Device to
                    operate as a USB device and allow connection to
                    a host PC system.
File
          CSV on SD Save the current waveform data as a text file
                    on an installed microSD card.
          CSV on USB Save the current waveform data as a text file
                    on an installed USB flash stick (if in USB host
                    mode - see the Setup group above).
          BMP on SD Save the current waveform display as a bitmap
                    on an installed microSD card.
          BMP on USB Save the current waveform display as a bitmap
                    on an installed USB flash stick (if in USB host
                    mode - see the Setup group above).
Help
          Help      Pressing "left" or "right" will show or hide the
                    screen showing oscilloscope connection help.
          Channel 1 Pressing "left" or "right" will cause the scale
                    and position for the channel 1 waveform to be set
                    such that the waveform is visible on the display.
          Channel 2 Pressing "left" or "right" will cause the scale
                    and position for the channel 2 waveform to be set
                    such that the waveform is visible on the display.
```

Oscilloscope Connections

The 8 pins immediately above the CSTN display panel offer connections for both channels of the oscilloscope and also two test signals that can be used to provide input in the absence of any other suitable signals. Each channel input must lie in the range -16.5V to +16.5V relative to the board ground allowing differences of up to 33V to be measured.

The connections are as follow where pin 1 is the leftmost pin, nearest the microSD card socket:

```
1    Test 1      A test signal connected to one side of the speaker on
                 the board.
2    Channel 1+  This is the positive connection for channel 1 of the
                 oscilloscope.
3    Channel 1-  This is the negative connection for channel 1 of the
                 oscilloscope.
4    Ground      This is connected to board ground.
5    Test 2      A test signal connected to the board Status LED which
                 is driven from PWM0.  This signal is configured to
                 provide a 1KHz square wave.
6    Channel 2+  This is the positive connection for channel 2 of the
                 oscilloscope.
7    Channel 2-  This is the negative connection for channel 2 of the
                 oscilloscope.
8    Ground      This is connected to board ground.
```

Triggering and Sample Rate Notes

The oscilloscope can sample at a maximum combined rate of 1M samples per second. When both channels are enabled, therefore, the maximum sample rate on each channel is 500K samples per second. For maximum resolution at the lowest timebases (maximum samples rates), disable channel 2 if it is not required. These sample rates give usable waveform capture for signals up to around 100KHz.

Trigger detection is performed in software during ADC interrupt handling. At the highest sampling rates, this interrupt service routine consumes almost all the available CPU cycles when searching

for trigger conditions. At these sample rates, if a trigger level is set which does not correspond to a voltage that is ever seen in the trigger channel signal, the user interface response can become sluggish. To combat this, the oscilloscope will abort any pending waveform capture operation if a key is pressed before the capture cycle as completed. This prevents the user interface from being locked out and allows the trigger level or type to be changed to values more appropriate for the signal being measured.

File Operations

Comma-separated-value or bitmap files representing the last waveform captured may be saved to either a microSD card or a USB flash drive. In each case, the files are written to the root directory of the microSD card or flash drive with filenames of the form "scopeXXX.csv" or "scopeXXX.bmp" where "XXX" represents the lowest, three digit, decimal number which offers a filename which does not already exist on the device.

Companion Application

A companion application, LMScope, which runs on WindowsXP and Vista PCs and the required device driver installer are available on the software CD and via download from the Luminary Micro web site at http://www.luminarymicro.com/products/software_updates.html. This application offers full control of the oscilloscope from the PC and allows waveform display and save to local hard disk.

SD card using FAT file system (sd_card)

This example application demonstrates reading a file system from an SD card. It makes use of FatFs, a FAT file system driver. It provides a simple command console via a serial port for issuing commands to view and navigate the file system on the SD card.

The first UART, which is connected to the FTDI virtual serial port on the Stellaris LM3S3748 Evaluation Board, is configured for 115,200 bits per second, and 8-n-1 mode. When the program is started a message will be printed to the terminal. Type "help" for command help.

For additional details about FatFs, see the following site: http://elm-chan.org/fsw/ff/00index_e.html

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own indicator on the display.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (connected to the FTDI virtual serial port on the evaluation board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

uDMA (udma_demo)

This example application demonstrates the use of the uDMA controller to transfer data between memory buffers, and to transfer data to and from a UART.

USB Generic Bulk Device (usb_dev_bulk)

This example provides a generic USB device offering simple bulk data transfer to and from the host. The device uses a vendor-specific class ID and supports a single bulk IN endpoint and a single bulk OUT endpoint. Data received from the host is assumed to be ASCII text and it is echoed back with the case of all alphabetic characters swapped.

A Windows INF file for the device is provided on the installation CD. This INF contains information required to install the WinUSB subsystem on WindowsXP and Vista PCs. WinUSB is a Windows subsystem allowing user mode applications to access the USB device without the need for a vendor-specific kernel mode driver. A sample Windows command-line application, usb_bulk_example, illustrating how to connect to and communicate with the bulk device is also provided as part of the Windows examples package on the installation CD or via download from http://www.luminarymicro.com/products/software_updates.html. Project files are included to allow the examples to be build using Microsoft VisualStudio 2005.

USB HID Keyboard Device (usb_dev_keyboard)

This example application turns the evaluation board into a USB keyboard supporting the Human Interface Device class. The CSTN display shows a virtual keyboard which can be navigated using the direction control button on the board. Pressing down on the button presses the highlighted key, sending its usage code and, if necessary, a shift modifier, to the USB host. The board status LED is used to indicate the current Caps Lock state and is updated in response to pressing the “Caps” key on the virtual keyboard or any other keyboard attached to the same USB host system.

The device implemented by this application also supports USB remote wakeup allowing it to request the host to reactivate a suspended bus. If the bus is suspended (as indicated on the application display), pressing the Select key will request a remote wakeup assuming the host has not specifically disabled such requests.

USB HID Mouse Device (usb_dev_mouse)

This example application turns the evaluation board into a USB mouse supporting the Human Interface Device class. Presses on the navigation control on the evaluation board are translated into mouse movement and button press messages in HID reports sent to the USB host allowing the evaluation board to control the mouse pointer on the host system.

USB Serial Device (usb_dev_serial)

This example application turns the evaluation kit into a virtual serial port when connected to the USB host system. The application supports the USB Communication Device Class, Abstract Control Model to redirect UART0 traffic to and from the USB host system. File usb_dev_serial_win2k.inf

may be used to install the example as a virtual COM port on a Windows2000 system. For WindowsXP or Vista, usb_dev_serial.inf should be used.

USB HID Keyboard Host (usb_host_keyboard)

This example application demonstrates how to support a USB keyboard attached to the evaluation kit board. The display will show if a keyboard is currently connected and the current state of the Caps Lock key on the keyboard that is connected on the bottom status area of the screen. Pressing any keys on the keyboard will cause them to be printed on the screen and to be sent out the UART at 115200 baud with no parity, 8 bits and 1 stop bit. Any keyboard that supports the USB HID bios protocol should work with this demo application.

USB HID Mouse Host (usb_host_mouse)

This example application demonstrates how to support a USB mouse attached to the evaluation kit board. The display will show if a mouse is currently connected and the current state of the buttons on the on the bottom status area of the screen. The main drawing area will show a mouse cursor that can be moved around in the main area of the screen. If the left mouse button is held while moving the mouse, the cursor will draw on the screen. A side effect of the application not being able to read the current state of the screen is that the cursor will erase anything it moves over while the left mouse button is not pressed.

USB Mass Storage Class Host (usb_host_msc)

This example application demonstrates how to connect a USB mass storage class device to the evaluation kit. When a device is detected, the application displays the contents of the file system and allows browsing using the buttons.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED is inverted so that it is easy to see that it is being fed, which occurs once every second.

44 EK-LM3S6965 Example Applications

44.1 Introduction

The EK-LM3S6965 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the OSRAM 128x64 4-bit gray-scale OLED graphical display on the Stellaris LM3S6965 Evaluation Kit boards.

These examples and display driver are for the EK-LM3S6965 Rev A boards, which utilize the 128x64 OSRAM display. The Rev A boards can be identified by looking on the back of the circuit board opposite the JTAG header. The board part number is located there and will end with an "A". If the board part number ends with a "C", then refer instead to the examples chapter for the EK-LM3S6965 Rev C Example Applications.

There is an IAR workspace file (`ek-lm3s6965.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`ek-lm3s6965-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`ek-lm3s6965.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/ek-lm3s6965` subdirectory of the peripheral driver library source distribution.

44.2 API Functions

Functions

- void OSRAM128x64x4Clear (void)
- void OSRAM128x64x4Disable (void)
- void OSRAM128x64x4DisplayOff (void)
- void OSRAM128x64x4DisplayOn (void)
- void OSRAM128x64x4Enable (unsigned long ulFrequency)
- void OSRAM128x64x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)
- void OSRAM128x64x4Init (unsigned long ulFrequency)

- void OSRAM128x64x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY, unsigned char ucLevel)

44.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

44.2.2 Function Documentation

44.2.2.1 OSRAM128x64x4Clear

Clears the OLED display.

Prototype:

```
void
OSRAM128x64x4Clear(void)
```

Description:

This function will clear the display RAM. All pixels in the display will be turned off.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:

None.

44.2.2.2 OSRAM128x64x4Disable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
OSRAM128x64x4Disable(void)
```

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:

None.

44.2.2.3 OSRAM128x64x4DisplayOff

Turns off the OLED display.

Prototype:

```
void
OSRAM128x64x4DisplayOff(void)
```

Description:

This function will turn off the OLED display. This will stop the scanning of the panel and turn off the on-chip DC-DC converter, preventing damage to the panel due to burn-in (it has similar characters to a CRT in this respect).

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:

None.

44.2.2.4 OSRAM128x64x4DisplayOn

Turns on the OLED display.

Prototype:

```
void
OSRAM128x64x4DisplayOn(void)
```

Description:

This function will turn on the OLED display, causing it to display the contents of its internal frame buffer.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:

None.

44.2.2.5 OSRAM128x64x4Enable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
OSRAM128x64x4Enable(unsigned long ulFrequency)
```

Parameters:

ulFrequency specifies the SSI Clock Frequency to be used.

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:

None.

44.2.2.6 OSRAM128x64x4ImageDraw

Displays an image on the OLED display.

Prototype:

```
void
OSRAM128x64x4ImageDraw(const unsigned char *pucImage,
                       unsigned long ulX,
                       unsigned long ulY,
                       unsigned long ulWidth,
                       unsigned long ulHeight)
```

Parameters:

pucImage is a pointer to the image data.

ulX is the horizontal position to display this image, specified in columns from the left edge of the display.

ulY is the vertical position to display this image, specified in rows from the top of the display.

ulWidth is the width of the image, specified in columns.

ulHeight is the height of the image, specified in rows.

Description:

This function will display a bitmap graphic on the display. Because of the format of the display RAM, the starting column (*ulX*) and the number of columns (*ulWidth*) must be an integer multiple of two.

The image data is organized with the first row of image data appearing left to right, followed immediately by the second row of image data. Each byte contains the data for two columns in the current row, with the leftmost column being contained in bits 7:4 and the rightmost column being contained in bits 3:0.

For example, an image six columns wide and seven scan lines tall would be arranged as follows (showing how the twenty one bytes of the image would appear on the display):

```
+-------------------+-------------------+-------------------+
|      Byte 0       |      Byte 1       |      Byte 2       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 3       |      Byte 4       |      Byte 5       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 6       |      Byte 7       |      Byte 8       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 9       |      Byte 10      |      Byte 11      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 12      |      Byte 13      |      Byte 14      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 15      |      Byte 16      |      Byte 17      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 18      |      Byte 19      |      Byte 20      |
```

```
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
```

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:
None.

44.2.2.7 OSRAM128x64x4Init

Initialize the OLED display.

Prototype:
```
void
OSRAM128x64x4Init(unsigned long ulFrequency)
```

Parameters:
ulFrequency specifies the SSI Clock Frequency to be used.

Description:
This function initializes the SSI interface to the OLED display and configures the SSD0323 controller on the panel.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Returns:
None.

44.2.2.8 OSRAM128x64x4StringDraw

Displays a string on the OLED display.

Prototype:
```
void
OSRAM128x64x4StringDraw(const char *pcStr,
                        unsigned long ulX,
                        unsigned long ulY,
                        unsigned char ucLevel)
```

Parameters:
pcStr is a pointer to the string to display.
ulX is the horizontal position to display the string, specified in columns from the left edge of the display.
ulY is the vertical position to display the string, specified in rows from the top edge of the display.
ucLevel is the 4-bit gray scale value to be used for displayed text.

Description:

This function will draw a string on the display. Only the ASCII characters between 32 (space) and 126 (tilde) are supported; other characters will result in random data being draw on the display (based on whatever appears before/after the font in memory). The font is mono-spaced, so characters such as “i” and “l” have more white space around them than characters such as “m” or “w”.

If the drawing of the string reaches the right edge of the display, no more characters will be drawn. Therefore, special care is not required to avoid supplying a string that is “too long” to display.

This function is contained in `osram128x64x4.c`, with `osram128x64x4.h` containing the API definition for use by applications.

Note:

Because the OLED display packs 2 pixels of data in a single byte, the parameter *ulX* must be an even column number (for example, 0, 2, 4, and so on).

Returns:

None.

44.3 Building Web Server File System Images

Control and configuration of applications running on an EL-LM3S6965 can be very conveniently implemented using an embedded HTTP server. This is illustrated in the enet_lwip example application.

The data sent to the client by the web server is generated using standard web development tools and stored into a normal directory on your development system (for example C:\DriverLib\boards\ek-lm3s6965\enet_lwip\fs). An application including the relevant file system drivers may chose to store its configuration web site on an SD card and serve it from there, in which case all that is required is to copy the required directory structure to the card and ensure that it is installed in the IDM module microSD slot. This approach frees an application from the limits imposed by the size of the microcontroller flash but makes the application dependent upon data that is outside its direct control. Assuming the site size is small enough to fit within the available flash, another, often better, method is to generate an image of the file system which is embedded within the application binary and accessed via internal file system calls. Both these approaches are supported by the enet_lwip example application.

To generate the internal file system image, two tools are provided. Each writes a C output file that contains an image of all the files contained within a subtree of the development system’s directory structure.

The first, makefsdata, is a Perl script which can be found in directory third_party/lwip-1.3.0/apps/httpserver_raw. This script takes 2 parameters - the name of the directory whose contents are to be included within the file system image, and the name of the C file that is to be written. Running this script from within the enet_lwip example application directory to create the file `lmi-fsdata.c`, the syntax would be:

```
perl ../../../third_party/lwip-1.3.0/apps/httpserver_raw/makefsdata fs
lmi-fsdata.c
```

If your development system does not have Perl installed, a Windows command line executable is also provided. This tool, makefsfile.exe, produces output files which are compatible with those

generated by makefsdata and offers a few additional features that may prove helpful in some circumstances. To generate the same output file as in the previous example, the syntax for running makefsfile would be:

```
..\makefsfile -i fs -o lmi-fsdata.c
```

By default, the file system image embeds the HTTP headers associated with each file in the file system image data itself. This is the default assumption of the lwIP web server implementation and is sensible if using an internal file system image containing a small number of files. If also serving files from a file system which does not embed the headers (for example the FAT file system on a microSD card as supported by the enet_lwip application) dynamic header generation must be used and internal file system images should be built using the -h option to makefsfile. In these cases, ensure that you also define `DYNAMIC_HTTP_HEADERS` in the `lwipopts.h` file to correctly configure the web server.

The -x option allows an "exclude file" to be specified. This exclude file contains the names of files and directories within the input directory tree that are to be skipped in the conversion process. If this option is not present, a default set of file excludes is used. This list contains typical source code control metadata directory names (".svn" and "CVS") and system files such as "thumbs.db". To see the default exclude list, run the tool with the -v option and look in the output.

Each file or directory name in the exclude file must be on a separate line within the file and each must be followed by a standard Windows ("\r\n") or Unix ("\n") line delimiter. The exclude list must contain individual file or directory names and my not include partial paths. For example `images_old` or `.svn` would be acceptable but `images_old/.svn` would not.

For a full list of command line options, run makefsfile with the "-?" option.

44.4 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

Ethernet with lwIP (enet_lwip)

This example application demonstrates the operation of the Stellaris Ethernet controller using the lwIP TCP/IP Stack. DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, AUTOIP will be used to obtain a link-local address. The address that is selected will be shown on the OLED display.

The file system code will first check to see if an SD card has been plugged into the microSD slot. If so, all file requests from the web server will be directed to the SD card. Otherwise, a default set of pages served up by an internal file system will be used.

For additional details on lwIP, refer to the lwIP web page at: http://www.sics.se/~adam/lwip/

Ethernet with lwIP (enet_lwip)

This example application demonstrates the operation of the Stellaris Ethernet controller using the lwIP TCP/IP Stack. DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, AUTOIP will be used to obtain a link-local address. The address that is selected will be shown on the OLED display.

A default set of pages will be served up by an internal file system and the httpd server.

The IEEE 1588 (PTP) software has been enabled in this code to synchronize the internal clock to a network master clock source.

Two methods of receive packet timestamping are implemented. The default mode uses the Stellaris hardware timestamp mechanism to capture Ethernet packet reception time using timer 3B. On parts which do not support hardware timestamping or if the application is started up with the Evaluation Kit "Select" button pressed, software time stamping is used.

By default, the system runs without any additional loading over and above that imposed by basic operation. To simulate a heavier load, the application includes a badly behaved timer interrupt handler which will merely steal CPU cycles each time it is invoked. To enable this loading, start the application while pressing the Evaluation Kit "Up" button. The amount of loading imposed by this interrupt handler can be tailored using labels defined in enet_ptpd.c

For additional details on lwIP, refer to the lwIP web page at: http://www.sics.se/~adam/lwip/

For additional details on the PTPd software, refer to the PTPd web page at: http://ptpd.sourceforge.net

Ethernet with uIP (enet_uip)

This example application demonstrates the operation of the Stellaris Ethernet controller using the uIP TCP/IP Stack. A basic web site is served over the Ethernet port, located at link local address 169.254.19.63. If a node on the network has already chosen this link local address, nothing is done by the application to choose another address and a conflict will occur. The web site displays a few lines of text, and a counter that increments each time the page is sent.

For additional details on uIP, refer to the uIP web page at: http://www.sics.se/~adam/uip/

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the select push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO.

Graphics Example (graphics)

A simple application that displays scrolling text on the top line of the OLED display, along with a 4-bit gray scale image.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the OLED and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the OLED; GPIO pins B0, B1 and B2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 440 Hz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

EK-LM3S6965 Quickstart Application (qs_ek-lm3s6965)

A game in which a blob-like character tries to find its way out of a maze. The character starts in the middle of the maze and must find the exit, which will always be located at one of the four corners of the maze. Once the exit to the maze is located, the character is placed into the middle of a new maze and must find the exit to that maze; this repeats endlessly.

The game is started by pressing the select push button on the right side of the board. During game play, the select push button will fire a bullet in the direction the character is currently facing, and the navigation push buttons on the left side of the board will cause the character to walk in the corresponding direction.

Populating the maze are a hundred spinning stars that mindlessly attack the character. Contact with one of these stars results in the game ending, but the stars go away when shot.

Score is accumulated for shooting the stars and for finding the exit to the maze. The game lasts for only one character, and the score is displayed on the virtual UART at 115,200, 8-N-1 during game play and will be displayed on the screen at the end of the game.

A small web site is provided by the game over the Ethernet port. DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, a static IP address will be used.

The DHCP timeout and the default static IP are easily configurable using macros. The address that is selected will be shown on the OLED display before the game starts. The web pages allow the entire game maze to be viewed, along with the character and stars; the display is generated by a Java applet that is downloaded from the game (therefore requiring that Java be installed in the web browser). The volume of the game music and sound effects can also be adjusted.

Since the OLED display on the evaluation board has burn-in characteristics similar to a CRT, the application also contains a screen saver. The screen saver will only become active if two minutes have passed without the user push button being pressed while waiting to start the game (that is, it will never come on during game play). Qix-style bouncing lines are drawn on the display by the screen saver.

After two minutes of running the screen saver, the display will be turned off and the user LED will blink. Either mode of screen saver (bouncing lines or blank display) will be exited by pressing the select push button. The select push button will then need to be pressed again to start the game.

SD card using FAT file system (sd_card)

This example application demonstrates reading a file system from an SD card. It makes use of FatFs, a FAT file system driver. It provides a simple command console via a serial port for issuing commands to view and navigate the file system on the SD card.

The first UART, which is connected to the FTDI virtual serial port on the Stellaris LM3S6965 Evaluation Board, is configured for 115,200 bits per second, and 8-n-1 mode. When the program is started a message will be printed to the terminal. Type "help" for command help.

For additional details about FatFs, see the following site: http://elm-chan.org/fsw/ff/00index_e.html

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own indicator on the display.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (connected to the FTDI virtual serial port on the evaluation board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED is inverted so that it is easy to see that it is being fed, which occurs once every second.

45 EK-LM3S6965 Rev C Example Applications

45.1 Introduction

The EK-LM3S6965 Rev C example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the RiTdisplay 128x96 4-bit gray-scale OLED graphical display on the Stellaris LM3S6965 Rev C Evaluation Kit boards.

These examples and display driver are for the EK-LM3S6965 Rev C boards, which utilize the 128x96 RiTdisplay display. The Rev C boards can be identified by looking on the back of the circuit board opposite the JTAG header. The board part number is located there and will end with an "C". If the board part number ends with a "A", then refer instead to the examples chapter for the EK-LM3S6965 Example Applications.

There is an IAR workspace file (`ek-lm3s6965_revc.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`ek-lm3s6965_revc-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`ek-lm3s6965_revc.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/ek-lm3s6965_revc` subdirectory of the peripheral driver library source distribution.

45.2 API Functions

Functions

- void RIT128x96x4Clear (void)
- void RIT128x96x4Disable (void)
- void RIT128x96x4DisplayOff (void)
- void RIT128x96x4DisplayOn (void)
- void RIT128x96x4Enable (unsigned long ulFrequency)
- void RIT128x96x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)
- void RIT128x96x4Init (unsigned long ulFrequency)

- void RIT128x96x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY, unsigned char ucLevel)

45.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

45.2.2 Function Documentation

45.2.2.1 RIT128x96x4Clear

Clears the OLED display.

Prototype:

```
void
RIT128x96x4Clear(void)
```

Description:

This function will clear the display RAM. All pixels in the display will be turned off.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

45.2.2.2 RIT128x96x4Disable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
RIT128x96x4Disable(void)
```

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

45.2.2.3 RIT128x96x4DisplayOff

Turns off the OLED display.

Prototype:

```
void
RIT128x96x4DisplayOff(void)
```

Description:

This function will turn off the OLED display. This will stop the scanning of the panel and turn off the on-chip DC-DC converter, preventing damage to the panel due to burn-in (it has similar characters to a CRT in this respect).

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

45.2.2.4 RIT128x96x4DisplayOn

Turns on the OLED display.

Prototype:

```
void
RIT128x96x4DisplayOn(void)
```

Description:

This function will turn on the OLED display, causing it to display the contents of its internal frame buffer.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

45.2.2.5 RIT128x96x4Enable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
RIT128x96x4Enable(unsigned long ulFrequency)
```

Parameters:

ulFrequency specifies the SSI Clock Frequency to be used.

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

45.2.2.6 RIT128x96x4ImageDraw

Displays an image on the OLED display.

Prototype:

```
void
RIT128x96x4ImageDraw(const unsigned char *pucImage,
                     unsigned long ulX,
                     unsigned long ulY,
                     unsigned long ulWidth,
                     unsigned long ulHeight)
```

Parameters:

pucImage is a pointer to the image data.

ulX is the horizontal position to display this image, specified in columns from the left edge of the display.

ulY is the vertical position to display this image, specified in rows from the top of the display.

ulWidth is the width of the image, specified in columns.

ulHeight is the height of the image, specified in rows.

Description:

This function will display a bitmap graphic on the display. Because of the format of the display RAM, the starting column (*ulX*) and the number of columns (*ulWidth*) must be an integer multiple of two.

The image data is organized with the first row of image data appearing left to right, followed immediately by the second row of image data. Each byte contains the data for two columns in the current row, with the leftmost column being contained in bits 7:4 and the rightmost column being contained in bits 3:0.

For example, an image six columns wide and seven scan lines tall would be arranged as follows (showing how the twenty one bytes of the image would appear on the display):

```
+-------------------+-------------------+-------------------+
|      Byte 0       |      Byte 1       |      Byte 2       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 3       |      Byte 4       |      Byte 5       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 6       |      Byte 7       |      Byte 8       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 9       |      Byte 10      |      Byte 11      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 12      |      Byte 13      |      Byte 14      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 15      |      Byte 16      |      Byte 17      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 18      |      Byte 19      |      Byte 20      |
```

```
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
```

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:
None.

45.2.2.7 RIT128x96x4Init

Initialize the OLED display.

Prototype:

```
void
RIT128x96x4Init(unsigned long ulFrequency)
```

Parameters:
ulFrequency specifies the SSI Clock Frequency to be used.

Description:
This function initializes the SSI interface to the OLED display and configures the SSD1329 controller on the panel.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:
None.

45.2.2.8 RIT128x96x4StringDraw

Displays a string on the OLED display.

Prototype:

```
void
RIT128x96x4StringDraw(const char *pcStr,
                      unsigned long ulX,
                      unsigned long ulY,
                      unsigned char ucLevel)
```

Parameters:
pcStr is a pointer to the string to display.
ulX is the horizontal position to display the string, specified in columns from the left edge of the display.
ulY is the vertical position to display the string, specified in rows from the top edge of the display.
ucLevel is the 4-bit gray scale value to be used for displayed text.

Description:

This function will draw a string on the display. Only the ASCII characters between 32 (space) and 126 (tilde) are supported; other characters will result in random data being draw on the display (based on whatever appears before/after the font in memory). The font is mono-spaced, so characters such as “i” and “l” have more white space around them than characters such as “m” or “w”.

If the drawing of the string reaches the right edge of the display, no more characters will be drawn. Therefore, special care is not required to avoid supplying a string that is “too long” to display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Note:

Because the OLED display packs 2 pixels of data in a single byte, the parameter *ulX* must be an even column number (for example, 0, 2, 4, and so on).

Returns:

None.

45.3 Building Web Server File System Images

Control and configuration of applications running on an EL-LM3S6965 can be very conveniently implemented using an embedded HTTP server. This is illustrated in the enet_lwip example application.

The data sent to the client by the web server is generated using standard web development tools and stored into a normal directory on your development system (for example C:\DriverLib\boards\ek-lm3s6965_revc\enet_lwip\fs). An application including the relevant file system drivers may chose to store its configuration web site on an SD card and serve it from there, in which case all that is required is to copy the required directory structure to the card and ensure that it is installed in the IDM module microSD slot. This approach frees an application from the limits imposed by the size of the microcontroller flash but makes the application dependent upon data that is outside its direct control. Assuming the site size is small enough to fit within the available flash, another, often better, method is to generate an image of the file system which is embedded within the application binary and accessed via internal file system calls. Both these approaches are supported by the enet_lwip example application.

To generate the internal file system image, two tools are provided. Each writes a C output file that contains an image of all the files contained within a subtree of the development system’s directory structure.

The first, makefsdata, is a Perl script which can be found in directory third_party/lwip-1.3.0/apps/httpserver_raw. This script takes 2 parameters - the name of the directory whose contents are to be included within the file system image, and the name of the C file that is to be written. Running this script from within the enet_lwip example application directory to create the file `lmi-fsdata.c`, the syntax would be:

```
perl ../../../third_party/lwip-1.3.0/apps/httpserver_raw/makefsdata fs
lmi-fsdata.c
```

If your development system does not have Perl installed, a Windows command line executable is also provided. This tool, makefsfile.exe, produces output files which are compatible with those

generated by makefsdata and offers a few additional features that may prove helpful in some circumstances. To generate the same output file as in the previous example, the syntax for running makefsfile would be:

```
..\makefsfile -i fs -o lmi-fsdata.c
```

By default, the file system image embeds the HTTP headers associated with each file in the file system image data itself. This is the default assumption of the lwIP web server implementation and is sensible if using an internal file system image containing a small number of files. If also serving files from a file system which does not embed the headers (for example the FAT file system on a microSD card as supported by the enet_lwip application) dynamic header generation must be used and internal file system images should be built using the -h option to makefsfile. In these cases, ensure that you also define `DYNAMIC_HTTP_HEADERS` in the `lwipopts.h` file to correctly configure the web server.

The -x option allows an "exclude file" to be specified. This exclude file contains the names of files and directories within the input directory tree that are to be skipped in the conversion process. If this option is not present, a default set of file excludes is used. This list contains typical source code control metadata directory names (".svn" and "CVS") and system files such as "thumbs.db". To see the default exclude list, run the tool with the -v option and look in the output.

Each file or directory name in the exclude file must be on a separate line within the file and each must be followed by a standard Windows ("\r\n") or Unix ("\n") line delimiter. The exclude list must contain individual file or directory names and my not include partial paths. For example `images_old` or `.svn` would be acceptable but `images_old/.svn` would not.

For a full list of command line options, run makefsfile with the "-?" option.

45.4 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

Ethernet with lwIP (enet_lwip)

This example application demonstrates the operation of the Stellaris Ethernet controller using the lwIP TCP/IP Stack. DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, AUTOIP will be used to obtain a link-local address. The address that is selected will be shown on the OLED display.

The file system code will first check to see if an SD card has been plugged into the microSD slot. If so, all file requests from the web server will be directed to the SD card. Otherwise, a default set of pages served up by an internal file system will be used.

For additional details on lwIP, refer to the lwIP web page at: http://www.sics.se/~adam/lwip/

Ethernet with lwIP (enet_lwip)

This example application demonstrates the operation of the Stellaris Ethernet controller using the lwIP TCP/IP Stack. DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, AUTOIP will be used to obtain a link-local address. The address that is selected will be shown on the OLED display.

A default set of pages will be served up by an internal file system and the httpd server.

The IEEE 1588 (PTP) software has been enabled in this code to synchronize the internal clock to a network master clock source.

Two methods of receive packet timestamping are implemented. The default mode uses the Stellaris hardware timestamp mechanism to capture Ethernet packet reception time using timer 3B. On parts which do not support hardware timestamping or if the application is started up with the Evaluation Kit "Select" button pressed, software time stamping is used.

By default, the system runs without any additional loading over and above that imposed by basic operation. To simulate a heavier load, the application includes a badly behaved timer interrupt handler which will merely steal CPU cycles each time it is invoked. To enable this loading, start the application while pressing the Evaluation Kit "Up" button. The amount of loading imposed by this interrupt handler can be tailored using labels defined in enet_ptpd.c

For additional details on lwIP, refer to the lwIP web page at: http://www.sics.se/~adam/lwip/

For additional details on the PTPd software, refer to the PTPd web page at: http://ptpd.sourceforge.net

Ethernet with uIP (enet_uip)

This example application demonstrates the operation of the Stellaris Ethernet controller using the uIP TCP/IP Stack. A basic web site is served over the Ethernet port, located at link local address 169.254.19.63. If a node on the network has already chosen this link local address, nothing is done by the application to choose another address and a conflict will occur. The web site displays a few lines of text, and a counter that increments each time the page is sent.

For additional details on uIP, refer to the uIP web page at: http://www.sics.se/~adam/uip/

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the select push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO.

Graphics Example (graphics)

A simple application that displays scrolling text on the top line of the OLED display, along with a 4-bit gray scale image.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the OLED and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the OLED; GPIO pins B0, B1 and B2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 440 Hz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

EK-LM3S6965 Rev C Quickstart Application (qs_ek-lm3s6965_revc)

A game in which a blob-like character tries to find its way out of a maze. The character starts in the middle of the maze and must find the exit, which will always be located at one of the four corners of the maze. Once the exit to the maze is located, the character is placed into the middle of a new maze and must find the exit to that maze; this repeats endlessly.

The game is started by pressing the select push button on the right side of the board. During game play, the select push button will fire a bullet in the direction the character is currently facing, and the navigation push buttons on the left side of the board will cause the character to walk in the corresponding direction.

Populating the maze are a hundred spinning stars that mindlessly attack the character. Contact with one of these stars results in the game ending, but the stars go away when shot.

Score is accumulated for shooting the stars and for finding the exit to the maze. The game lasts for only one character, and the score is displayed on the virtual UART at 115,200, 8-N-1 during game play and will be displayed on the screen at the end of the game.

A small web site is provided by the game over the Ethernet port. DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, a static IP address will be used.

The DHCP timeout and the default static IP are easily configurable using macros. The address that is selected will be shown on the OLED display before the game starts. The web pages allow the entire game maze to be viewed, along with the character and stars; the display is generated by a Java applet that is downloaded from the game (therefore requiring that Java be installed in the web browser). The volume of the game music and sound effects can also be adjusted.

Since the OLED display on the evaluation board has burn-in characteristics similar to a CRT, the application also contains a screen saver. The screen saver will only become active if two minutes have passed without the user push button being pressed while waiting to start the game (that is, it will never come on during game play). Qix-style bouncing lines are drawn on the display by the screen saver.

After two minutes of running the screen saver, the display will be turned off and the user LED will blink. Either mode of screen saver (bouncing lines or blank display) will be exited by pressing the select push button. The select push button will then need to be pressed again to start the game.

SD card using FAT file system (sd_card)

This example application demonstrates reading a file system from an SD card. It makes use of FatFs, a FAT file system driver. It provides a simple command console via a serial port for issuing commands to view and navigate the file system on the SD card.

The first UART, which is connected to the FTDI virtual serial port on the Stellaris LM3S6965 Evaluation Board, is configured for 115,200 bits per second, and 8-n-1 mode. When the program is started a message will be printed to the terminal. Type "help" for command help.

For additional details about FatFs, see the following site: http://elm-chan.org/fsw/ff/00index_e.html

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own indicator on the display.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (connected to the FTDI virtual serial port on the evaluation board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED is inverted so that it is easy to see that it is being fed, which occurs once every second.

46 EK-LM3S811 Example Applications

46.1 Introduction

The EK-LM3S811 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the OSRAM 96x16 OLED graphical display on the Stellaris LM3S811 Evaluation Kit board.

There is an IAR workspace file (`ek-lm3s811.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`ek-lm3s811-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`ek-lm3s811.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/ek-lm3s811` subdirectory of the peripheral driver library source distribution.

46.2 API Functions

Functions

- void OSRAM96x16x1Clear (void)
- void OSRAM96x16x1DisplayOff (void)
- void OSRAM96x16x1DisplayOn (void)
- void OSRAM96x16x1ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)
- void OSRAM96x16x1Init (tBoolean bFast)
- void OSRAM96x16x1StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY)

46.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

There are macros to map the old function names for the OSRAM driver to the new function names (such as OSRAMInit to OSRAM96x16x1Init). The new names are more descriptive of the panel type in use; the macros for the old names are provided for backward compatibility.

46.2.2 Function Documentation

46.2.2.1 OSRAM96x16x1Clear

Clears the OLED display.

Prototype:

```
void
OSRAM96x16x1Clear(void)
```

Description:

This function will clear the display. All pixels in the display will be turned off.

This function is contained in `osram96x16x1.c`, with `osram96x16x1.h` containing the API definition for use by applications.

Returns:

None.

46.2.2.2 OSRAM96x16x1DisplayOff

Turns off the OLED display.

Prototype:

```
void
OSRAM96x16x1DisplayOff(void)
```

Description:

This function will turn off the OLED display. This will stop the scanning of the panel and turn off the on-chip DC-DC converter, preventing damage to the panel due to burn-in (it has similar characters to a CRT in this respect).

This function is contained in `osram96x16x1.c`, with `osram96x16x1.h` containing the API definition for use by applications.

Returns:

None.

46.2.2.3 OSRAM96x16x1DisplayOn

Turns on the OLED display.

Prototype:

```
void
OSRAM96x16x1DisplayOn(void)
```

Description:

This function will turn on the OLED display, causing it to display the contents of its internal frame buffer.

This function is contained in `osram96x16x1.c`, with `osram96x16x1.h` containing the API definition for use by applications.

Returns:

None.

46.2.2.4 OSRAM96x16x1ImageDraw

Displays an image on the OLED display.

Prototype:

```
void
OSRAM96x16x1ImageDraw(const unsigned char *pucImage,
                      unsigned long ulX,
                      unsigned long ulY,
                      unsigned long ulWidth,
                      unsigned long ulHeight)
```

Parameters:

pucImage is a pointer to the image data.

ulX is the horizontal position to display this image, specified in columns from the left edge of the display.

ulY is the vertical position to display this image, specified in eight scan line blocks from the top of the display (that is, only 0 and 1 are valid).

ulWidth is the width of the image, specified in columns.

ulHeight is the height of the image, specified in eight row blocks (that is, only 1 and 2 are valid).

Description:

This function will display a bitmap graphic on the display. The image to be displayed must be a multiple of eight scan lines high (that is, one row) and will be drawn at a vertical position that is a multiple of eight scan lines (that is, scan line zero or scan line eight, corresponding to row zero or row one).

The image data is organized with the first row of image data appearing left to right, followed immediately by the second row of image data. Each byte contains the data for the eight scan lines of the column, with the top scan line being in the least significant bit of the byte and the bottom scan line being in the most significant bit of the byte.

For example, an image four columns wide and sixteen scan lines tall would be arranged as follows (showing how the eight bytes of the image would appear on the display):

```
+-------+  +-------+  +-------+  +-------+
|   | 0 |  |   | 0 |  |   | 0 |  |   | 0 |
| B | 1 |  | B | 1 |  | B | 1 |  | B | 1 |
| y | 2 |  | y | 2 |  | y | 2 |  | y | 2 |
| t | 3 |  | t | 3 |  | t | 3 |  | t | 3 |
| e | 4 |  | e | 4 |  | e | 4 |  | e | 4 |
|   | 5 |  |   | 5 |  |   | 5 |  |   | 5 |
| 0 | 6 |  | 1 | 6 |  | 2 | 6 |  | 3 | 6 |
|   | 7 |  |   | 7 |  |   | 7 |  |   | 7 |
```

```
+-------+  +-------+  +-------+  +-------+

+-------+  +-------+  +-------+  +-------+
|   | 0 |  |   | 0 |  |   | 0 |  |   | 0 |
| B | 1 |  | B | 1 |  | B | 1 |  | B | 1 |
| y | 2 |  | y | 2 |  | y | 2 |  | y | 2 |
| t | 3 |  | t | 3 |  | t | 3 |  | t | 3 |
| e | 4 |  | e | 4 |  | e | 4 |  | e | 4 |
|   | 5 |  |   | 5 |  |   | 5 |  |   | 5 |
| 4 | 6 |  | 5 | 6 |  | 6 | 6 |  | 7 | 6 |
|   | 7 |  |   | 7 |  |   | 7 |  |   | 7 |
+-------+  +-------+  +-------+  +-------+
```

This function is contained in `osram96x16x1.c`, with `osram96x16x1.h` containing the API definition for use by applications.

Returns:
None.

46.2.2.5 OSRAM96x16x1Init

Initialize the OLED display.

Prototype:

```
void
OSRAM96x16x1Init(tBoolean bFast)
```

Parameters:
bFast is a boolean that is *true* if the I2C interface should be run at 400 kbps and *false* if it should be run at 100 kbps.

Description:
This function initializes the I2C interface to the OLED display and configures the SSD0303 controller on the panel.

This function is contained in `osram96x16x1.c`, with `osram96x16x1.h` containing the API definition for use by applications.

Returns:
None.

46.2.2.6 OSRAM96x16x1StringDraw

Displays a string on the OLED display.

Prototype:

```
void
OSRAM96x16x1StringDraw(const char *pcStr,
                       unsigned long ulX,
                       unsigned long ulY)
```

Parameters:
pcStr is a pointer to the string to display.

ulX is the horizontal position to display the string, specified in columns from the left edge of the display.

ulY is the vertical position to display the string, specified in eight scan line blocks from the top of the display (that is, only 0 and 1 are valid).

Description:

This function will draw a string on the display. Only the ASCII characters between 32 (space) and 126 (tilde) are supported; other characters will result in random data being draw on the display (based on whatever appears before/after the font in memory). The font is mono-spaced, so characters such as "i" and "l" have more white space around them than characters such as "m" or "w".

If the drawing of the string reaches the right edge of the display, no more characters will be drawn. Therefore, special care is not required to avoid supplying a string that is "too long" to display.

This function is contained in `osram96x16x1.c`, with `osram96x16x1.h` containing the API definition for use by applications.

Returns:

None.

46.3 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the user push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO. Note that because of errata in Rev Bx and Rev C0 of Sandstorm-class Stellaris microcontrollers, JTAG and SWD will not function if PB7 is configured as a GPIO. This errata is fixed in Rev C2 of Sandstorm-class Stellaris microcontrollers.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the LCD and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the LCD; GPIO pins D0 through D2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the

off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 50 kHz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

EK-LM3S811 Quickstart Application (qs_ek-lm3s811)

A game in which a ship is navigated through an endless tunnel. The potentiometer is used to move the ship up and down, and the user push button is used to fire a missile to destroy obstacles in the tunnel. Score accumulates for survival and for destroying obstacles. The game lasts for only one ship; the score is displayed on the virtual UART at 115,200, 8-N-1 during game play and will be displayed on the screen at the end of the game.

Since the OLED display on the evaluation board has burn-in characteristics similar to a CRT, the application also contains a screen saver. The screen saver will only become active if two minutes have passed without the user push button being pressed while waiting to start the game (that is, it will never come on during game play). An implementation of the Game of Life is run with a field of random data as the seed value.

After two minutes of running the screen saver, the display will be turned off and the user LED will blink. Either mode of screen saver (Game of Life or blank display) will be exited by pressing the user push button. The button will then need to be pressed again to start the game.

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own indicator on the display.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (connected to the FTDI virtual serial port on the Stellaris LM3S811 Evaluation Board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED connected to port C5 is inverted so that it is easy to see that it is being fed, which occurs once every second.

47 EK-LM3S8962 Example Applications

47.1 Introduction

The EK-LM3S8962 example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is a board specific driver for the RiTdisplay 128x96 4-bit gray-scale OLED graphical display on the Stellaris LM3S8962 Evaluation Kit board.

There is an IAR workspace file (`ek-lm3s8962.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`ek-lm3s8962-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`ek-lm3s8962.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/ek-lm3s8962` subdirectory of the peripheral driver library source distribution.

47.2 API Functions

Functions

- void RIT128x96x4Clear (void)
- void RIT128x96x4Disable (void)
- void RIT128x96x4DisplayOff (void)
- void RIT128x96x4DisplayOn (void)
- void RIT128x96x4Enable (unsigned long ulFrequency)
- void RIT128x96x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)
- void RIT128x96x4Init (unsigned long ulFrequency)
- void RIT128x96x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY, unsigned char ucLevel)

47.2.1 Detailed Description

Each API specifies the source file that contains it and the header file that provides the prototype for application use.

47.2.2 Function Documentation

47.2.2.1 RIT128x96x4Clear

Clears the OLED display.

Prototype:

```
void
RIT128x96x4Clear(void)
```

Description:

This function will clear the display RAM. All pixels in the display will be turned off.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

47.2.2.2 RIT128x96x4Disable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
RIT128x96x4Disable(void)
```

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

47.2.2.3 RIT128x96x4DisplayOff

Turns off the OLED display.

Prototype:

```
void
RIT128x96x4DisplayOff(void)
```

Description:

This function will turn off the OLED display. This will stop the scanning of the panel and turn off the on-chip DC-DC converter, preventing damage to the panel due to burn-in (it has similar characters to a CRT in this respect).

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

47.2.2.4 RIT128x96x4DisplayOn

Turns on the OLED display.

Prototype:

```
void
RIT128x96x4DisplayOn(void)
```

Description:

This function will turn on the OLED display, causing it to display the contents of its internal frame buffer.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

47.2.2.5 RIT128x96x4Enable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
RIT128x96x4Enable(unsigned long ulFrequency)
```

Parameters:

ulFrequency specifies the SSI Clock Frequency to be used.

Description:

This function initializes the SSI interface to the OLED display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:

None.

47.2.2.6 RIT128x96x4ImageDraw

Displays an image on the OLED display.

Prototype:

```
void
RIT128x96x4ImageDraw(const unsigned char *pucImage,
                     unsigned long ulX,
                     unsigned long ulY,
                     unsigned long ulWidth,
                     unsigned long ulHeight)
```

Parameters:

pucImage is a pointer to the image data.

ulX is the horizontal position to display this image, specified in columns from the left edge of the display.

ulY is the vertical position to display this image, specified in rows from the top of the display.

ulWidth is the width of the image, specified in columns.

ulHeight is the height of the image, specified in rows.

Description:

This function will display a bitmap graphic on the display. Because of the format of the display RAM, the starting column (*ulX*) and the number of columns (*ulWidth*) must be an integer multiple of two.

The image data is organized with the first row of image data appearing left to right, followed immediately by the second row of image data. Each byte contains the data for two columns in the current row, with the leftmost column being contained in bits 7:4 and the rightmost column being contained in bits 3:0.

For example, an image six columns wide and seven scan lines tall would be arranged as follows (showing how the twenty one bytes of the image would appear on the display):

```
+-------------------+-------------------+-------------------+
|      Byte 0       |      Byte 1       |      Byte 2       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 3       |      Byte 4       |      Byte 5       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 6       |      Byte 7       |      Byte 8       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 9       |      Byte 10      |      Byte 11      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 12      |      Byte 13      |      Byte 14      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 15      |      Byte 16      |      Byte 17      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 18      |      Byte 19      |      Byte 20      |
```

```
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
```

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:
None.

47.2.2.7 RIT128x96x4Init

Initialize the OLED display.

Prototype:

```
void
RIT128x96x4Init(unsigned long ulFrequency)
```

Parameters:
ulFrequency specifies the SSI Clock Frequency to be used.

Description:
This function initializes the SSI interface to the OLED display and configures the SSD1329 controller on the panel.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Returns:
None.

47.2.2.8 RIT128x96x4StringDraw

Displays a string on the OLED display.

Prototype:

```
void
RIT128x96x4StringDraw(const char *pcStr,
                      unsigned long ulX,
                      unsigned long ulY,
                      unsigned char ucLevel)
```

Parameters:
pcStr is a pointer to the string to display.
ulX is the horizontal position to display the string, specified in columns from the left edge of the display.
ulY is the vertical position to display the string, specified in rows from the top edge of the display.
ucLevel is the 4-bit gray scale value to be used for displayed text.

Description:

This function will draw a string on the display. Only the ASCII characters between 32 (space) and 126 (tilde) are supported; other characters will result in random data being draw on the display (based on whatever appears before/after the font in memory). The font is mono-spaced, so characters such as “i” and “l” have more white space around them than characters such as “m” or “w”.

If the drawing of the string reaches the right edge of the display, no more characters will be drawn. Therefore, special care is not required to avoid supplying a string that is “too long” to display.

This function is contained in `rit128x96x4.c`, with `rit128x96x4.h` containing the API definition for use by applications.

Note:

Because the OLED display packs 2 pixels of data in a single byte, the parameter *ulX* must be an even column number (for example, 0, 2, 4, and so on).

Returns:

None.

47.3 Building Web Server File System Images

Control and configuration of applications running on an EL-LM3S8962 can be very conveniently implemented using an embedded HTTP server. This is illustrated in the enet_lwip example application.

The data sent to the client by the web server is generated using standard web development tools and stored into a normal directory on your development system (for example C:\DriverLib\boards\ek-lm3s8962\enet_lwip\fs). An application including the relevant file system drivers may chose to store its configuration web site on an SD card and serve it from there, in which case all that is required is to copy the required directory structure to the card and ensure that it is installed in the IDM module microSD slot. This approach frees an application from the limits imposed by the size of the microcontroller flash but makes the application dependent upon data that is outside its direct control. Assuming the site size is small enough to fit within the available flash, another, often better, method is to generate an image of the file system which is embedded within the application binary and accessed via internal file system calls. Both these approaches are supported by the enet_lwip example application.

To generate the internal file system image, two tools are provided. Each writes a C output file that contains an image of all the files contained within a subtree of the development system’s directory structure.

The first, makefsdata, is a Perl script which can be found in directory third_party/lwip-1.3.0/apps/httpserver_raw. This script takes 2 parameters - the name of the directory whose contents are to be included within the file system image, and the name of the C file that is to be written. Running this script from within the enet_lwip example application directory to create the file `lmi-fsdata.c`, the syntax would be:

```
perl ../../../third_party/lwip-1.3.0/apps/httpserver_raw/makefsdata fs
lmi-fsdata.c
```

If your development system does not have Perl installed, a Windows command line executable is also provided. This tool, makefsfile.exe, produces output files which are compatible with those

generated by makefsdata and offers a few additional features that may prove helpful in some circumstances. To generate the same output file as in the previous example, the syntax for running makefsfile would be:

```
..\makefsfile -i fs -o lmi-fsdata.c
```

By default, the file system image embeds the HTTP headers associated with each file in the file system image data itself. This is the default assumption of the lwIP web server implementation and is sensible if using an internal file system image containing a small number of files. If also serving files from a file system which does not embed the headers (for example the FAT file system on a microSD card as supported by the enet_lwip application) dynamic header generation must be used and internal file system images should be built using the -h option to makefsfile. In these cases, ensure that you also define `DYNAMIC_HTTP_HEADERS` in the `lwipopts.h` file to correctly configure the web server.

The -x option allows an "exclude file" to be specified. This exclude file contains the names of files and directories within the input directory tree that are to be skipped in the conversion process. If this option is not present, a default set of file excludes is used. This list contains typical source code control metadata directory names (".svn" and "CVS") and system files such as "thumbs.db". To see the default exclude list, run the tool with the -v option and look in the output.

Each file or directory name in the exclude file must be on a separate line within the file and each must be followed by a standard Windows ("\r\n") or Unix ("\n") line delimiter. The exclude list must contain individual file or directory names and my not include partial paths. For example `images_old` or `.svn` would be acceptable but `images_old/.svn` would not.

For a full list of command line options, run makefsfile with the "-?" option.

47.4 Examples

Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

Blinky (blinky)

A very simple example that blinks the on-board LED.

Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

CAN Device Board LED Application (can_device_led)

This simple application uses the two buttons on the board as a light switch. When the “up” button is pressed the status LED will turn on. When the “down” button is pressed, the status LED will turn off.

CAN Device Board Quickstart Application (can_device_qs)

This application uses the CAN controller to communicate with the evaluation board that is running the example game. It receives messages over CAN to turn on, turn off, or to pulse the LED on the device board. It also sends CAN messages when either of the up and down buttons are pressed or released.

Ethernet with lwIP (enet_lwip)

This example application demonstrates the operation of the Stellaris Ethernet controller using the lwIP TCP/IP Stack. DHCP is used to obtain an Ethernet address. If DHCP times out without

obtaining an address, AUTOIP will be used to obtain a link-local address. The address that is selected will be shown on the OLED display.

The file system code will first check to see if an SD card has been plugged into the microSD slot. If so, all file requests from the web server will be directed to the SD card. Otherwise, a default set of pages served up by an internal file system will be used.

For additional details on lwIP, refer to the lwIP web page at: http://www.sics.se/~adam/lwip/

Ethernet with lwIP (enet_lwip)

This example application demonstrates the operation of the Stellaris Ethernet controller using the lwIP TCP/IP Stack. DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, AUTOIP will be used to obtain a link-local address. The address that is selected will be shown on the OLED display.

A default set of pages will be served up by an internal file system and the httpd server.

The IEEE 1588 (PTP) software has been enabled in this code to synchronize the internal clock to a network master clock source.

Two methods of receive packet timestamping are implemented. The default mode uses the Stellaris hardware timestamp mechanism to capture Ethernet packet reception time using timer 3B. On parts which do not support hardware timestamping or if the application is started up with the Evaluation Kit "Select" button pressed, software time stamping is used.

By default, the system runs without any additional loading over and above that imposed by basic operation. To simulate a heavier load, the application includes a badly behaved timer interrupt handler which will merely steal CPU cycles each time it is invoked. To enable this loading, start the application while pressing the Evaluation Kit "Up" button. The amount of loading imposed by this interrupt handler can be tailored using labels defined in enet_ptpd.c

For additional details on lwIP, refer to the lwIP web page at: http://www.sics.se/~adam/lwip/

For additional details on the PTPd software, refer to the PTPd web page at: http://ptpd.sourceforge.net

Ethernet with uIP (enet_uip)

This example application demonstrates the operation of the Stellaris Ethernet controller using the uIP TCP/IP Stack. A basic web site is served over the Ethernet port, located at link local address 169.254.19.63. If a node on the network has already chosen this link local address, nothing is done by the application to choose another address and a conflict will occur. The web site displays a few lines of text, and a counter that increments each time the page is sent.

For additional details on uIP, refer to the uIP web page at: http://www.sics.se/~adam/uip/

GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the select push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO.

Graphics Example (graphics)

A simple application that displays scrolling text on the top line of the OLED display, along with a 4-bit gray scale image.

Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the OLED and is a starting point for more complicated applications.

Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the OLED; GPIO pins B0, B1 and B2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 440 Hz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

EK-LM3S8962 Quickstart Application (qs_ek-lm3s8962)

A game in which a blob-like character tries to find its way out of a maze. The character starts in the middle of the maze and must find the exit, which will always be located at one of the four corners of the maze. Once the exit to the maze is located, the character is placed into the middle of a new maze and must find the exit to that maze; this repeats endlessly.

The game is started by pressing the select push button on the right side of the board. During game play, the select push button will fire a bullet in the direction the character is currently facing, and the navigation push buttons on the left side of the board will cause the character to walk in the corresponding direction.

Populating the maze are a hundred spinning stars that mindlessly attack the character. Contact with one of these stars results in the game ending, but the stars go away when shot.

Score is accumulated for shooting the stars and for finding the exit to the maze. The game lasts for only one character, and the score is displayed on the virtual UART at 115,200, 8-N-1 during game play and will be displayed on the screen at the end of the game.

A small web site is provided by the game over the Ethernet port. DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, a static IP address will be used. The DHCP timeout and the default static IP are easily configurable using macros. The address that is selected will be shown on the OLED display before the game starts. The web pages allow the entire game maze to be viewed, along with the character and stars; the display is generated by a Java applet that is downloaded from the game (therefore requiring that Java be installed in the web browser). The volume of the game music and sound effects can also be adjusted.

If the CAN device board is attached and is running the can_device_qs application, the volume of the music and sound effects can be adjusted over CAN with the two push buttons on the target board. The LED on the CAN device board will track the state of the LED on the main board via CAN messages. The operation of the game will not be affected by the absence of the CAN device board.

Since the OLED display on the evaluation board has burn-in characteristics similar to a CRT, the application also contains a screen saver. The screen saver will only become active if two minutes have passed without the user push button being pressed while waiting to start the game (that is, it will never come on during game play). Qix-style bouncing lines are drawn on the display by the screen saver.

After two minutes of running the screen saver, the display will be turned off and the user LED will blink. Either mode of screen saver (bouncing lines or blank display) will be exited by pressing the select push button. The select push button will then need to be pressed again to start the game.

SD card using FAT file system (sd_card)

This example application demonstrates reading a file system from an SD card. It makes use of FatFs, a FAT file system driver. It provides a simple command console via a serial port for issuing commands to view and navigate the file system on the SD card.

The first UART, which is connected to the FTDI virtual serial port on the Stellaris LM3S6965 Evaluation Board, is configured for 115,200 bits per second, and 8-n-1 mode. When the program is started a message will be printed to the terminal. Type “help” for command help.

For additional details about FatFs, see the following site: http://elm-chan.org/fsw/ff/00index_e.html

Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own indicator on the display.

UART (uart_echo)

This example application utilizes the UART to echo text. The first UART (connected to the FTDI virtual serial port on the evaluation board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED is inverted so that it is easy to see that it is being fed, which occurs once every second.

48 RDK-IDM Example Applications

48.1 Introduction

The RDK-IDM example applications show the capabilities of the Intelligent Display Module, the peripheral driver library, and the graphics library. These applications are intended for demonstration and as a starting point for new applications.

In addition to the graphics library display driver for the TFT display, there are board specific drivers for the analog input channels, relay output, sound output, and touch screen. A wrapper is also provided to simplify the initialization and operation of the lwIP TCP/IP stack.

The Intelligent Display Module comes in two variants: the MDL-IDM which supports Power over Ethernet (PoE) and the MDL-IDM28 which does not support PoE. The two modules are otherwise identical, and the software described in this chapter works the same on both modules.

There is an IAR workspace file (`rdk-idm.eww`) that contains the peripheral driver library and graphics library projects, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`rdk-idm-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`rdk-idm.mpw`) that contains the peripheral driver library and graphics library projects, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

48.1.1 Analog Input Driver

There are four analog input channels which can sense from 0 to 3 Volts in 1024 individual, evenly spaced steps. The analog input driver will sample these channels every millisecond and will call application-supplied callback functions when the value is above or below a set value, and when it crosses a set value (subject to hystersis) in either direction. Each channel can be independently configured, and can have individual callbacks for each event.

The analog input driver utilizes sample sequence 2 of the ADC and timer 0 subtimer A (shared with the touch screen driver).

48.1.2 Display Driver

In addition to providing the tDisplay structure required by the graphics library, the display driver also provides APIs for initializing the display, turning on the backlight, and turning off the backlight.

The display driver can be configured for four different orientations:

- Portrait, which is the default orientation of the display driver, the preferred orientation of the display, and the orientation used by all of the example applications. Portrait mode provides a 240x320 display and is selected by defining **PORTRAIT**, or by not defining an orientation, when building the display driver.
- Landscape, which is the screen rotated 90 degrees counter-clockwise (where the flex connector is on the right side of the display). Landscape mode provides a 320x240 display and is selected by defining **LANDSCAPE** when building the display driver.
- Portrait flip, which is the screen rotated 180 degrees (where the flex connector is on the top of the display). Portrait flip mode provides a 240x320 display and is selected by defining **PORTRAIT_FLIP** when building the display driver.
- Landscape flip, which is the screen rotated 90 degrees clockwise (where the flex connector is on the left side of the display). Landscape flip mode provides a 320x240 display and is selected by defining **LANDSCAPE_FLIP** when building the display driver.

The orientation used depends on the requirements of the application and the desired viewing angle of the display. The panel itself has a 6 o'clock viewing angle, so when used in portrait flip mode it has a 12 o'clock viewing angle. Similarly, it has a 3 o'clock viewing angle in landscape mode and a 9 o'clock viewing angle in landscape flip mode. Viewing the display from an angle other than the viewing angle will result in distortion of the displayed colors.

48.1.3 Relay Output Driver

The relay output has three pins; a common contact, a normally closed contact, and a normally opened contact. By enabling the relay output, the normally closed contact is opened and the normally opened contact is closed. When the relay output is disabled, it reverts back to its normal state.

The relay output driver provides a means of enabling and disabling the relay. The application is responsible for any sequencing, such as enabling the relay, delaying for one second, and then disabling the relay.

48.1.4 Sound Output Driver

The sound output provides a means of producing simple tones using a square wave drive. The sound output driver allows the output frequency to be changed, along with the volume of the sound. There is also a method for creating simple songs or sound effects by specifying a sequence of frequencies and the times at which they should be output.

When the PWM output goes high, the speaker will start to travel from its resting position to its fully deflected position. Since this takes time (on the order of 50 uS), it is possible to turn off the PWM output before it has reached the extent of its travel. Doing so will result in less air being displaced by the moving speaker cone, which results in a reduced volume. This technique is utilizes to provide a simple volume control.

Since a timer PWM output is used to drive the speaker, the maximum divide that is available is 65535. When running the processor (and therefore the timer) at 50 MHz, this equates to a minimum audio frequency of approximately 762.95 Hz. Lower audio frequencies are possible if the processor clock rate is lowered, though this will degrade the performance of the entire system (this will be most noticable in the update rate on the display).

The sound output driver utilizes timer 2 (both subtimer A and subtimer B).

48.1.5 Touch Screen Driver

The touch screen is a pair of resistive layers on the surface of the display. One layer has connection points at the top and bottom of the screen, and the other layer has connection points at the left and right of the screen. When the screen is touched, the two layers make contact and electricity can flow between them.

The horizontal position of a touch can be found by applying positive voltage to the right side of the horizontal layer and negative voltage to to the left side. When not driving the top and bottom of the vertical layer, the voltage potential on that layer will be proportional to the horizontal distance across the screen of the press, which can be measured with an ADC channel. By reversing these connections, the vertical position can also be measured. When the screen is not being touched, there will be no voltage on the non-powered layer.

By monitoring the voltage on each layer when the other layer is appropriately driven, touches and releases on the screen, as well as movements of the touch, can be detected and reported.

In order to read the current voltage on the two layers and also drive the appropriate voltages onto the layers, each side of each layer is connected to both a GPIO and an ADC channel. The GPIO is used to drive the node to a particular voltage, and when the GPIO is configured as an input, the corresponding ADC channel can be used to read the layer's voltage.

The touch screen is sampled every millisecond, with four samples required to properly read both the X and Y position. Therefore, 250 X/Y sample pairs are captured every second.

Like the display driver, the touch screen driver operates in the same four orientations (selected in the same manner). Default calibrations are provided for using the touch screen in each orientation; the calibrate application can be used to determine new calibration values if necessary.

The touch screen driver utilizes sample sequence 3 of the ADC and timer 0 subtimer A (shared with the analog input driver).

The touch screen driver makes use of calibration parameters determined using the "calibrate" example application. The theory behind these parameters is explained by Carlos E. Videles in the June 2002 issue of Embedded Systems Design. It can be found online at http://www.embedded.com/story/OEG20020529S0046.

48.2 Analog Input API Functions

Functions

- void AnalogCallbackSetAbove (unsigned long ulChannel, tAnalogCallback *pfnOnAbove)
- void AnalogCallbackSetBelow (unsigned long ulChannel, tAnalogCallback *pfnOnBelow)

- void AnalogCallbackSetFallingEdge (unsigned long ulChannel, tAnalogCallback *pfnOnFallingEdge)
- void AnalogCallbackSetRisingEdge (unsigned long ulChannel, tAnalogCallback *pfnOnRisingEdge)
- void AnalogInit (void)
- void AnalogIntHandler (void)
- void AnalogLevelSet (unsigned long ulChannel, unsigned short usLevel, char cHysteresis)

48.2.1 Detailed Description

These functions are contained in `analog.c`, with `analog.h` containing the API definitions for use by applications.

48.2.2 Function Documentation

48.2.2.1 AnalogCallbackSetAbove

Sets the function to be called when the analog input is above the trigger level.

Prototype:

```
void
AnalogCallbackSetAbove(unsigned long ulChannel,
                       tAnalogCallback *pfnOnAbove)
```

Parameters:

ulChannel is the channel to modify.

pfnOnAbove is a pointer to the function to be called whenever the analog input is above the trigger level.

Description:

This function sets the function that should be called whenever the analog input is above the trigger level (in other words, while the analog input is above the trigger level, the callback will be called every millisecond). Specifying a function address of 0 will cancel a previous callback function (meaning that no function will be called when the analog input is above the trigger level).

Returns:

None.

48.2.2.2 AnalogCallbackSetBelow

Sets the function to be called when the analog input is below the trigger level.

Prototype:

```
void
AnalogCallbackSetBelow(unsigned long ulChannel,
                       tAnalogCallback *pfnOnBelow)
```

Parameters:

ulChannel is the channel to modify.

pfnOnBelow is a pointer to the function to be called whenever the analog input is below the trigger level.

Description:

This function sets the function that should be called whenever the analog input is below the trigger level (in other words, while the analog input is below the trigger level, the callback will be called every millisecond). Specifying a function address of 0 will cancel a previous callback function (meaning that no function will be called when the analog input is below the trigger level).

Returns:

None.

48.2.2.3 AnalogCallbackSetFallingEdge

Sets the function to be called when the analog input transitions from above to below the trigger level.

Prototype:

```
void
AnalogCallbackSetFallingEdge(unsigned long ulChannel,
                             tAnalogCallback *pfnOnFallingEdge)
```

Parameters:

ulChannel is the channel to modify.

pfnOnFallingEdge is a pointer to the function to be called when the analog input transitions from above to below the trigger level.

Description:

This function sets the function that should be called whenever the analog input transitions from above to below the trigger level. Specifying a function address of 0 will cancel a previous callback function (meaning that no function will be called when the analog input transitions from above to below the trigger level).

Returns:

None.

48.2.2.4 AnalogCallbackSetRisingEdge

Sets the function to be called when the analog input transitions from below to above the trigger level.

Prototype:

```
void
AnalogCallbackSetRisingEdge(unsigned long ulChannel,
                            tAnalogCallback *pfnOnRisingEdge)
```

Parameters:

ulChannel is the channel to modify.

pfnOnRisingEdge is a pointer to the function to be called when the analog input transitions from below to above the trigger level.

Description:
This function sets the function that should be called whenever the analog input transitions from below to above the trigger level. Specifying a function address of 0 will cancel a previous callback function (meaning that no function will be called when the analog input transitions from below to above the trigger level).

Returns:
None.

48.2.2.5 AnalogInit

Initializes the analog input driver.

Prototype:
```
void
AnalogInit(void)
```

Description:
This function initializes the analog input driver, starting the sampling process and disabling all channel callbacks. Once called, the ADC2 interrupt will be asserted periodically; the AnalogIntHandler() function should be called in response to this interrupt. It is the application's responsibility to install AnalogIntHandler() in the application's vector table.

Returns:
None.

48.2.2.6 AnalogIntHandler

Handles the ADC sample sequence two interrupt.

Prototype:
```
void
AnalogIntHandler(void)
```

Description:
This function is called when the ADC sample sequence two generates an interrupt. It will read the new ADC readings, perform debouncing on the analog inputs, and call the appropriate callbacks based on the new readings.

Returns:
None.

48.2.2.7 AnalogLevelSet

Sets the trigger level for an analog channel.

Prototype:

```
void
AnalogLevelSet(unsigned long ulChannel,
               unsigned short usLevel,
               char cHysteresis)
```

Parameters:

ulChannel is the channel to modify.

usLevel is the trigger level for this channel.

cHysteresis is the hysteresis to apply to the trigger level for this channel.

Description:

This function sets the trigger level and hysteresis for an analog input channel. The hysteresis allows for filtering of noise on the analog input. The actual level to transition from "below" the trigger level to "above" the trigger level is the trigger level plus the hysteresis. Similarly, the actual level to transition from "above" the trigger level to "below" the trigger level is the trigger level minus the hysteresis.

Returns:

None.

48.3 Display Driver API Functions

Functions

- void Formike240x320x16_ILI9320BacklightOff (void)
- void Formike240x320x16_ILI9320BacklightOn (void)
- unsigned short Formike240x320x16_ILI9320ControllerIdGet (void)
- void Formike240x320x16_ILI9320Init (void)

Variables

- const tDisplay g_sFormike240x320x16_ILI9320

48.3.1 Detailed Description

These functions are contained in `formike240x320x16_ili9320.c`, with `formike240x320x16_ili9320.h` containing the API definitions for use by applications.

48.3.2 Function Documentation

48.3.2.1 Formike240x320x16_ILI9320BacklightOff

Turns off the backlight.

Prototype:

```
void
Formike240x320x16_ILI9320BacklightOff(void)
```

Description:

This function turns off the backlight on the display.

Returns:

None.

48.3.2.2 Formike240x320x16_ILI9320BacklightOn

Turns on the backlight.

Prototype:

```
void
Formike240x320x16_ILI9320BacklightOn(void)
```

Description:

This function turns on the backlight on the display.

Returns:

None.

48.3.2.3 Formike240x320x16_ILI9320ControllerIdGet

Determines whether an ILI9320 or ILI9325 controller is present.

Prototype:

```
unsigned short
Formike240x320x16_ILI9320ControllerIdGet(void)
```

Description:

This function queries the ID of the display controller in use and returns it to the caller. This driver supports both ILI9320 and ILI9325. These are very similar but the sense of the long display axis is reversed in the Formike KWH028Q02-F05 using an ILI9325 controller and this information is needed by the touchscreen driver to provide correct touch coordinate information.

Returns:

Returns 0x9320 if an ILI9320 controller is in use or 0x9325 if an ILI9325 is present.

48.3.2.4 Formike240x320x16_ILI9320Init

Initializes the display driver.

Prototype:

```
void
Formike240x320x16_ILI9320Init(void)
```

Description:
This function initializes the ILI9320 or ILI9325 display controller on the panel, preparing it to display data.

Returns:
None.

48.3.3 Variable Documentation

48.3.3.1 g_sFormike240x320x16_ILI9320

Definition:
```
const tDisplay g_sFormike240x320x16_ILI9320
```

Description:
The display structure that describes the driver for the Formike Electronic KWH028Q02-F03 TFT panel with an ILI9320 controller.

48.4 Relay Output API Functions

Functions

- void RelayDisable (void)
- void RelayEnable (void)
- void RelayInit (void)

48.4.1 Detailed Description

These functions are contained in `relay.c`, with `relay.h` containing the API definitions for use by applications.

48.4.2 Function Documentation

48.4.2.1 RelayDisable

Disable the relay output.

Prototype:
```
void
RelayDisable(void)
```

Description:
This function disables the relay output. This causes the relay to become de-energized, putting it into its default state (in other words, the normally open connect is opened and the normally closed contact is closed).

Returns:
None.

48.4.2.2 RelayEnable

Enables the relay output.

Prototype:
```
void
RelayEnable(void)
```

Description:
This function enables the relay output. This causes the relay to become energized, putting it into the non-default state (in other words, the normally open contact is closed and the normally closed contact is opened).

Returns:
None.

48.4.2.3 RelayInit

Initializes the relay output.

Prototype:
```
void
RelayInit(void)
```

Description:
This function initializes the relay output, preparing it to control the relay. The relay is started in the disabled state (in other words, de-energized).

Returns:
None.

48.5 Sound Output API Functions

Functions

- void SoundDisable (void)
- void SoundEnable (void)
- void SoundFrequencySet (unsigned long ulFrequency)
- void SoundInit (void)
- void SoundIntHandler (void)
- void SoundPlay (const unsigned short *pusSong, unsigned long ulLength)
- void SoundVolumeDown (unsigned long ulPercent)
- unsigned char SoundVolumeGet (void)
- void SoundVolumeSet (unsigned long ulPercent)
- void SoundVolumeUp (unsigned long ulPercent)

48.5.1 Detailed Description

These functions are contained in `sound.c`, with `sound.h` containing the API definitions for use by applications.

48.5.2 Function Documentation

48.5.2.1 SoundDisable

Disables the sound output.

Prototype:

```
void
SoundDisable(void)
```

Description:

This function disables the sound output, muting the speaker and cancelling any playback that may be in progress.

Returns:

None.

48.5.2.2 SoundEnable

Enables the sound output.

Prototype:

```
void
SoundEnable(void)
```

Description:

This function enables the sound output, preparing it to play music or sound effects.

Returns:

None.

48.5.2.3 SoundFrequencySet

Sets the sound output frequency.

Prototype:

```
void
SoundFrequencySet(unsigned long ulFrequency)
```

Parameters:

ulFrequency is the desired sound output frequency.

Description:
This function sets the frequency of the output sound. This change will take effect immediately and will remain in effect until changed (either explicitly by another call or implicitly by the playback of a sound).

Returns:
None.

48.5.2.4 SoundInit

Initializes the sound output.

Prototype:
```
void
SoundInit(void)
```

Description:
This function prepares the sound driver to play songs or sound effects. It must be called before any other sound functions. The sound driver uses timer 2 subtimer A to produce the PWM output, and timer 2 subtimer B to be the time base for the playback of sound effects. It is the responsility of the application to ensure that SoundIntHandler() is called when the timer 2 subtimer B interrupt occurs (typically by placing a pointer to this function in the appropriate location in the processor's vector table).

Returns:
None.

48.5.2.5 SoundIntHandler

Handles the sound timer interrupt.

Prototype:
```
void
SoundIntHandler(void)
```

Description:
This function provides periodic updates to the PWM output in order to produce a sound effect. It is called when the timer 2 subtimer B interrupt occurs.

Returns:
None.

48.5.2.6 SoundPlay

Starts playback of a song.

Prototype:
```
void
SoundPlay(const unsigned short *pusSong,
          unsigned long ulLength)
```

Parameters:

pusSong is a pointer to the song data structure.

ulLength is the length of the song data structure in bytes.

Description:

This function starts the playback of a song or sound effect. If a song or sound effect is already being played, its playback is cancelled and the new song is started.

Returns:

None.

48.5.2.7 SoundVolumeDown

Decreases the volume.

Prototype:

```
void
SoundVolumeDown(unsigned long ulPercent)
```

Parameters:

ulPercent is the amount to decrease the volume, specified as a percentage between 0% (silence) and 100% (full volume), inclusive.

Description:

This function adjusts the audio output down by the specified percentage. The adjusted volume will not go below 0% (silence).

Returns:

None.

48.5.2.8 SoundVolumeGet

Returns the current volume level.

Prototype:

```
unsigned char
SoundVolumeGet(void)
```

Description:

This function returns the current volume, specified as a percentage between 0% (silence) and 100% (full volume), inclusive.

Returns:

Returns the current volume.

48.5.2.9 SoundVolumeSet

Sets the volume of the music/sound effect playback.

Prototype:

```
void
SoundVolumeSet(unsigned long ulPercent)
```

Parameters:

ulPercent is the volume percentage, which must be between 0% (silence) and 100% (full volume), inclusive.

Description:

This function sets the volume of the sound output to a value between silence (0%) and full volume (100%).

Returns:

None.

48.5.2.10 SoundVolumeUp

Increases the volume.

Prototype:

```
void
SoundVolumeUp(unsigned long ulPercent)
```

Parameters:

ulPercent is the amount to increase the volume, specified as a percentage between 0% (silence) and 100% (full volume), inclusive.

Description:

This function adjusts the audio output up by the specified percentage. The adjusted volume will not go above 100% (full volume).

Returns:

None.

48.6 Touch Screen API Functions

Functions

- void TouchScreenCallbackSet (long (*pfnCallback)(unsigned long ulMessage, long lX, long lY))
- void TouchScreenInit (void)
- void TouchScreenIntHandler (void)

48.6.1 Detailed Description

These functions are contained in `touch.c`, with `touch.h` containing the API definitions for use by applications.

48.6.2 Function Documentation

48.6.2.1 TouchScreenCallbackSet

Sets the callback function for touch screen events.

Prototype:

```
void
TouchScreenCallbackSet(long (*long)(unsigned ulMessage, long lX, long
lY) pfnCallback)
```

Parameters:

pfnCallback is a pointer to the function to be called when touch screen events occur.

Description:

This function sets the address of the function to be called when touch screen events occur. The events that are recognized are the screen being touched ("pen down"), the touch position moving while the screen is touched ("pen move"), and the screen no longer being touched ("pen up").

Returns:

None.

48.6.2.2 TouchScreenInit

Initializes the touch screen driver.

Prototype:

```
void
TouchScreenInit(void)
```

Description:

This function initializes the touch screen driver, beginning the process of reading from the touch screen. This driver uses the following hardware resources:

- ADC sample sequence 3
- Timer 0 subtimer A

Returns:

None.

48.6.2.3 TouchScreenIntHandler

Handles the ADC interrupt for the touch screen.

Prototype:

```
void
TouchScreenIntHandler(void)
```

Description:

This function is called when the ADC sequence that samples the touch screen has completed its acquisition. The touch screen state machine is advanced and the acquired ADC sample is processed appropriately.

It is the responsibility of the application using the touch screen driver to ensure that this function is installed in the interrupt vector table for the ADC3 interrupt.

Returns:

None.

48.7 Boot Loader and Firmware Update

Each of the RDK-IDM example applications is configured to operate alongside the boot loader to facilitate firmware update operations over Ethernet. The LM Flash Programmer, available for download from http://www.luminarymicro.com/products/software_updates.html, may be used to replace the main application image in each case. Software is included within each application to listen for "magic packets" from LM Flash Programmer indicating that a user is requesting a firmware update and, if such a packet is received, to pass control back to the bootloader to initiate the BOOTP/TFTP firmware update process. Note that the "magic packet" functionality is available in versions of LM Flash Programmer with build numbers greater than 560 (the build number can be found in the tool's "About..." box or on the overview page of the application help file). Older versions of LM Flash Programmer supporting Ethernet operation may be used to update applications which offer a manual method of initiating the firmware update but not those which rely solely upon the "magic packet" trigger.

The Ethernet-enabled boot loader image is built from the boot_eth directory under the main boards/rdk-idm directory. Typically, it will not be necessary to flash this image unless you have made changes to the boot loader source code. Replacing the boot loader image requires the use of a hardware JTAG/SWD debugger or a Luminary Evaluation Kit board configured as a JTAG/SWD pass-through adapter.

48.8 Building Web Server File System Images

Control and configuration of applications running on an RDK-IDM can be very conveniently implemented using an embedded HTTP server. This is illustrated in the qs-keypad example application which uses a web server to change the door lock access code.

The data sent to the client by the web server is generated using standard web development tools and stored into a normal directory on your development system (for example C:\DriverLib\boards\rdk-idm\qs-keypad\html). An application including the relevant file system drivers may chose to store its configuration web site on an SD card and serve it from there, in which case all that is required is to copy the required directory structure to the card and ensure that it is installed in the IDM module microSD slot. This approach frees an application from the limits imposed by the size of the microcontroller flash but makes the application dependent upon data that is outside its direct control. Assuming the site size is small enough to fit within the available flash, another, often better, method is to generate an image of the file system which is embedded within the application binary and accessed via internal file system calls. This is the approach adopted by the qs-keypad example application.

To generate the internal file system image, two tools are provided. Each writes a C output file that contains an image of all the files contained within a subtree of the development system's directory structure.

The first, makefsdata, is a Perl script which can be found in directory third_party/lwip-1.3.0/apps/httpserver_raw. This script takes 2 parameters - the name of the directory whose contents are to be included within the file system image, and the name of the C file that is to be written. Running this script from within the qs-keypad example application directory to create the file `fsdata-qs.c`, the syntax would be:

```
perl ../../../third_party/lwip-1.3.0/apps/httpserver_raw/makefsdata html
fsdata-qs.c
```

If your development system does not have Perl installed, a Windows command line executable is also provided. This tool, makefsfile.exe, produces output files which are compatible with those generated by makefsdata and offers a few additional features that may prove helpful in some circumstances. To generate the same output file as in the previous example, the syntax for running makefsfile would be:

```
..\makefsfile -i html -o fsdata-qs.c
```

By default, the file system image embeds the HTTP headers associated with each file in the file system image data itself. This is the default assumption of the lwIP web server implementation and is sensible if using an internal file system image containing a small number of files. If also serving files from a file system which does not embed the headers (for example the FAT file system on a microSD card) dynamic header generation must be used and internal file system images should be built using the -h option to makefsfile. In these cases, ensure that you also define `DYNAMIC_HTTP_HEADERS` in the `lwipopts.h` file to correctly configure the web server.

The -x option allows an "exclude file" to be specified. This exclude file contains the names of files and directories within the input directory tree that are to be skipped in the conversion process. If this option is not present, a default set of file excludes is used. This list contains typical source code control metadata directory names (".svn" and "CVS") and system files such as "thumbs.db". To see the default exclude list, run the tool with the -v option and look in the output.

Each file or directory name in the exclude file must be on a separate line within the file and each must be followed by a standard Windows ("\r\n") or Unix ("\n") line delimiter. The exclude list must contain individual file or directory names and my not include partial paths. For example `images_old` or `.svn` would be acceptable but `images_old/.svn` would not.

For a full list of command line options, run makefsfile with the "-?" option.

48.9 Examples

All of these examples reside in the `boards/rdk-idm` subdirectory of the peripheral driver library source distribution.

BLDC RDK Control (bldc_ctrl)

This application provides a simple GUI for controlling a BLDC RDK board. The motor can be started and stopped, the target speed can be adjusted, and the current speed can be monitored.

The target speed up and down buttons utilize the auto-repeat capability of the push button widget.

For example, pressing the up button will increase the target speed by 100 rpm. Holding it for more than 0.5 seconds will commence the auto-repeat, at which point the target speed will increase by 100 rpm every 1/10th of a second. The same behavior occurs on the down button.

Upon startup, the application will attempt to contact a DHCP server to get an IP address. If a DHCP server can not be contacted, it will instead use the IP address 169.254.19.70 without performing any ARP checks to see if it is already in use. Once the IP address is determined, it will initiate a connection to a BLDC RDK board at IP address 169.254.89.71. While attempting to contact the DHCP server and the BLDC RDK board, the target speed will display as a set of bouncing dots.

The push buttons will not operate until a connection to a BLDC RDK board has been established.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer. This feature is available in versions of LM Flash Programmer with build numbers greater than 560.

Boot Loader (boot_eth)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses Ethernet to load an application.

Calibration for the Touch Screen (calibrate)

The raw sample interface of the touch screen driver is used to compute the calibration matrix required to convert raw samples into screen X/Y positions. The produced calibration matrix can be inserted into the touch screen driver to map the raw samples into screen coordinates.

The touch screen calibration is performed according to the algorithm described by Carlos E. Videles in the June 2002 issue of Embedded Systems Design. It can be found online at http://www.embedded.com/story/OEG20020529S0046.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer. This feature is available in versions of LM Flash Programmer with build numbers greater than 560.

Graphics Library Demonstration (grlib_demo)

This application provides a demonstration of the capabilities of the Stellaris Graphics Library. A series of panels show different features of the library. For each panel, the bottom provides a forward and back button (when appropriate), along with a brief description of the contents of the panel.

The first panel provides some introductory text and basic instructions for operation of the application.

The second panel shows the available drawing primitives: lines, circles, rectangles, strings, and images.

The third panel shows the canvas widget, which provides a general drawing surface within the widget heirarchy. A text, image, and application-drawn canvas are displayed.

The fourth panel shows the check box widget, which provides a means of toggling the state of an item. Four check boxes are provided, with each having a red "LED" to the right. The state of the LED tracks the state of the check box via an application callback.

The fifth panel shows the container widget, which provides a grouping construct typically used for radio buttons. Containers with a title, a centered title, and no title are displayed.

The sixth panel shows the push button widget. Two columns of push buttons are provided; the appearance of each column is the same but the left column does not utilize auto-repeat while the right column does. Each push button has a red "LED" to its left, which is toggled via an application callback each time the push button is pressed.

The seventh panel shows the radio button widget. Two groups of radio buttons are displayed, the first using text and the second using images for the selection value. Each radio button has a red "LED" to its right, which tracks the selection state of the radio buttons via an application callback. Only one radio button from each group can be selected at a time, though the radio buttons in each group operate independently.

The eighth panel shows the slider widget. Six sliders constructed using the various supported style options are shown. The slider value callback is used to update two widgets to reflect the values reported by sliders. A canvas widget in the top right of the display tracks the value of the red and green image-based slider to its left and the text of the grey slider on the left side of the panel is update to show its own value. The rightmost slider is configured as an indicator which tracks the state of the upper slider and ignores user input.

The final panel provides instructions and information necessary to update the board firmware via ethernet using the LM Flash Programmer application. When using a version of LM Flash Programmer with a build number greater than 560, software updates will occur automatically without user intervention being required in the application. If using an earlier version of LM Flash Programmer which does not send the "magic packet" signalling an update request, the "Update" button on the final screen may be pressed to transfer control to the boot loader in preparation for a firmware download.

Hello World (hello)

A very simple "hello world" example. It simply displays "Hello World!" on the display and is a starting point for more complicated applications.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request "magic packet" from LM Flash Programmer. This feature is available in versions of LM Flash Programmer with build numbers greater than 560.

Quickstart Security Keypad (qs-keypad)

This application provides a security keypad to allow access to a door. The relay output is momentarily toggled upon entry of the access code to activate an electric door strike, unlocking the door.

The screen is divided into three parts; the Luminary Micro banner across the top, a hint across the

bottom, and the main application area in the middle (which is the only portion that should appear if this application is used for a real door access system). The hints provide an on-screen guide to what the application is expecting at any given time.

Upon startup, the screen is blank and the hint says to touch the screen. Pressing the screen will bring up the keypad, which is randomized as an added security measure (so that an observer can not "steal" the access code by simply looking at the relative positions of the button presses). The current access code is provide in the hint at the bottom of the screen (which is clearly not secure).

If an incorrect access code is entered ("#" ends the code entry), then the screen will go blank and wait for another access attempt. If the correct access code is entered, the relay will be toggled for a few seconds (as indicated by the hint at the bottom stating that the door is open) and the screen will go blank. Once the door is closed again, the screen can be touched again to repeat the process.

The UART is used to output a log of events. Each event in the log is time stamped, with the arbitrary date of February 26, 2008 at 14:00 UT (universal time) being the starting time when the application is run. The following events are logged:

- The start of the application
- The access code being changed
- Access being granted (correct access code being entered)
- Access being denied (incorrect access code being entered)
- The door being relocked after access has been granted

A simple web server is provided to allow the access code to be changed. The Ethernet interface will attempt to contact a DHCP server, and if it is unable to acquire a DHCP address it will instead use the IP address 169.254.19.70 without performing any ARP checks to see if it is already in use. The web page shows the current access code and provides a form for updating the access code.

If a micro-SD card is present, the access code will be stored in a file called "key.txt" in the root directory. This file is written whenever the access code is changed, and is read at startup to initialize the access code. If a micro-SD card is not present, or the "key.txt" file does not exist, the access code defaults to 6918.

If "$**$" is entered on the numeric keypad, the application provides a demonstration of the Stellaris Graphics Library with various panels showing the available widget types and graphics primitives. Navigate between the panels using buttons marked "+" and "-" at the bottom of the screen and return to keypad mode by pressing the "X" buttons which appear when you are on either the first or last demonstration panel.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated via the remote update request "magic packet" from LM Flash Programmer. If using

Note that remote firmware update signalling is only supported in versions of LM Flash Programmer with build numbers greater than 560. If using an earlier version of LM Flash Programmer which does not send the "magic packet" signalling an update request, an update may be initiated by entering "$*$0" on the application's numeric keypad.

Scribble Pad (scribble)

The scribble pad provides a drawing area on the screen. Touching the screen will draw onto the drawing area using a selection of fundamental colors (in other words, the seven colors produced by

the three color channels being either fully on or fully off). Each time the screen is touched to start a new drawing, the drawing area is erased and the next color is selected.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request "magic packet" from LM Flash Programmer. This feature is available in versions of LM Flash Programmer with build numbers greater than 560.

SD card using FAT file system (sd_card)

This example application demonstrates reading a file system from an SD card. It makes use of FatFs, a FAT file system driver. It provides a simple widget-based console on the display and also a UART-based command line for viewing and navigating the file system on the SD card.

For additional details about FatFs, see the following site: http://elm-chan.org/fsw/ff/00index_e.html

UART1, which is connected to the 3 pin header on the underside of the IDM RDK board (J2), is configured for 115,200 bits per second, and 8-n-1 mode. When the program is started a message will be printed to the terminal. Type "help" for command help.

To connect the IDM RDK board's UART to a 9 pin PC serial port, use a standard male to female DB9 serial cable and connect TXD (J2 pin 1, nearest the SD card socket) to pin 2 of the male serial cable connector, RXD (J2 pin 2, the center pin) to pin 3 of the serial connector and GND (J2 pin 3) to pin 5 of the serial connector.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request "magic packet" from LM Flash Programmer. This feature is available in versions of LM Flash Programmer with build numbers greater than 560.

49 RDK-S2E Example Applications

49.1 Introduction

The RDK-S2E example applications show the capabilities of the Serial to Ethernet Module and the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is an IAR workspace file (`rdk-s2e.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is also an equivalent IAR workspace file (`rdk-s2e-ewarm4.eww`) for use with Embedded Workbench version 4.42a.

There is a Keil multi-project workspace file (`rdk-s2e.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

The file `ser2ent.c` contains the main application entry point. The various modules are initialized, including the serial port driver, the telnet driver, the Universal Plug and Play driver, along with the configuration web server.

To provide periodic processing that is required by lwIP, the system tick timer is programmed and the interrupt service routine for this timer is contained here.

The lwIP Abstraction Library also provides for a host callback routine that can be configured as a periodic callback run in the lwIP context, avoiding reentrancy issues that exist with lwIP. The host callback routine defined here provides support for the telnet and upnp modules.

49.2 Configuration API Functions

Data Structures

- tStringMap

Defines

- DEFAULT_CGI_RESPONSE

- FIRMWARE_UPDATE_RESPONSE
- IP_UPDATE_RESPONSE
- MAX_VARIABLE_NAME_LEN
- MISC_PAGE_URI
- NUM_CONFIG_CGI_URIS
- NUM_CONFIG_SSI_TAGS
- PARAM_ERROR_RESPONSE

Functions

- void ConfigInit (void)
- void ConfigLoad (void)
- void ConfigLoadFactory (void)
- void ConfigSave (void)
- void ConfigWebInit (void)

Variables

- tBoolean g_bChangeIPAddress
- tBoolean g_bStartBootloader
- const tConfigParameters * g_psDefaultParameters
- const tConfigParameters *const g_psFactoryParameters
- tConfigParameters g_sParameters
- const unsigned short g_usFirmwareVersion

49.2.1 Detailed Description

The configuration module defines and manages the global configuration parameter block, as well providing an abstraction layer for the non-volatile storage of this parameter block.

These functions are contained in `config.c`, with `config.h` containing the API definitions for use by applications.

49.2.2 Data Structure Documentation

49.2.2.1 tStringMap

Definition:

```
typedef struct
{
    const char *pcString;
    unsigned char ucId;
}
tStringMap
```

Members:

pcString A human readable string related to the identifier found in the ucId field.

ucId An identifier value associated with the string held in the pcString field.

Description:

Structure used in mapping numeric IDs to human-readable strings.

49.2.3 Define Documentation

49.2.3.1 DEFAULT_CGI_RESPONSE

Definition:

```
#define DEFAULT_CGI_RESPONSE
```

Description:

The file sent back to the browser by default following completion of any of our CGI handlers.

49.2.3.2 FIRMWARE_UPDATE_RESPONSE

Definition:

```
#define FIRMWARE_UPDATE_RESPONSE
```

Description:

The file sent back to the browser to signal that the bootloader is being entered to perform a software update.

49.2.3.3 IP_UPDATE_RESPONSE

Definition:

```
#define IP_UPDATE_RESPONSE
```

Description:

The file sent back to the browser to signal that the IP address of the device is about to change and that the web server is no longer operating.

49.2.3.4 MAX_VARIABLE_NAME_LEN

Definition:

```
#define MAX_VARIABLE_NAME_LEN
```

Description:

The maximum length of any HTML form variable name used in this application.

49.2.3.5 MISC_PAGE_URI

Definition:

```
#define MISC_PAGE_URI
```

Description:

The URI of the “Miscellaneous Settings” page offered by the web server.

49.2.3.6 NUM_CONFIG_CGI_URIS

Definition:

```
#define NUM_CONFIG_CGI_URIS
```

Description:

The number of individual CGI URIs that are used by our configuration web pages.

49.2.3.7 NUM_CONFIG_SSI_TAGS

Definition:

```
#define NUM_CONFIG_SSI_TAGS
```

Description:

The number of individual SSI tags that the HTTPD server can expect to find in our configuration pages.

49.2.3.8 PARAM_ERROR_RESPONSE

Definition:

```
#define PARAM_ERROR_RESPONSE
```

Description:

The file sent back to the browser in cases where a parameter error is detected by one of the CGI handlers. This should only happen if someone tries to access the CGI directly via the broswer command line and doesn’t enter all the required parameters alongside the URI.

49.2.4 Function Documentation

49.2.4.1 ConfigInit

Initializes the configuration parameter block.

Prototype:

```
void
ConfigInit(void)
```

Description:
This function initializes the configuration parameter block. If the version number of the parameter block stored in flash is older than the current revision, new paramters will be set to default values as needed.

Returns:
None.

49.2.4.2 ConfigLoad

Loads the S2E parameter block from flash.

Prototype:
```
void
ConfigLoad(void)
```

Description:
This function is called to load the most recently saved parameter block from flash.

Returns:
None.

49.2.4.3 ConfigLoadFactory

Loads the S2E parameter block from factory-default table.

Prototype:
```
void
ConfigLoadFactory(void)
```

Description:
This function is called to load the factory default parameter block.

Returns:
None.

49.2.4.4 ConfigSave

Saves the S2E parameter block to flash.

Prototype:
```
void
ConfigSave(void)
```

Description:
This function is called to save the current S2E configuration parameter block to flash memory.

Returns:
None.

49.2.4.5 ConfigWebInit

Configures HTTPD server SSI and CGI capabilities for our configuration forms.

Prototype:

```
void
ConfigWebInit(void)
```

Description:

This function informs the HTTPD server of the server-side-include tags that we will be processing and the special URLs that are used for CGI processing for the web-based configuration forms.

Returns:

None.

49.2.5 Variable Documentation

49.2.5.1 g_bChangeIPAddress

Definition:

```
tBoolean g_bChangeIPAddress
```

Description:

A flag to the main loop indicating that it should update the IP address after a short delay (to allow us to send a suitable page back to the web browser telling it the address has changed).

49.2.5.2 g_bStartBootloader

Definition:

```
tBoolean g_bStartBootloader
```

Description:

A flag to the main loop indicating that it should enter the bootloader and perform a firmware update.

49.2.5.3 g_psDefaultParameters

Definition:

```
const tConfigParameters *g_psDefaultParameters
```

Description:

This structure instance points to the most recently saved parameter block in flash. It can be considered the default set of parameters.

49.2.5.4 g_psFactoryParameters

Definition:

```
const tConfigParameters *const g_psFactoryParameters
```

Description:

This structure instance points to the factory default set of parameters in flash memory.

49.2.5.5 g_sParameters

Definition:

```
tConfigParameters g_sParameters
```

Description:

This structure instance contains the run-time set of configuration parameters for S2E module. This is the active parameter set and may contain changes that are not to be committed to flash.

49.2.5.6 g_usFirmwareVersion

Definition:

```
const unsigned short g_usFirmwareVersion
```

Description:

The version of the firmware. Changing this value will make it much more difficult for Luminary Micro support personnel to determine the firmware in use when trying to provide assistance; it should only be changed after careful consideration.

49.3 File System API Functions

Functions

- void fs_close (struct fs_file *file)
- fs_file * fs_open (char *name)
- int fs_read (struct fs_file *file, char *buffer, int count)

49.3.1 Detailed Description

This set of functions provides an interface to the built-in flash file system used by the lwIP-based web server. In addition to providing handles to the normal flash-based files, "special" files can be handled in such a way to provide dynamic content to the web client.

The original data for the web server file system can be found in the `fs` directory. These files are converted into a single C file, `fsdata-s2e.c` which is then included in `fs_s2e.c`. The file system data file `fsdata-s2e.c` can be generated in one of two ways depending upon the tools you have installed. If cygwin and Perl are available, the script `makefsdata.pl`, found in the third_party/lwip-1.3.0/apps/httpserver_raw directory below the DriverLib installation directory can

be run. Alternatively, if Perl is not installed, the Windows command line utility, `makefsfile.exe` found in the boards/rdk-s2e directory can be run.

Starting in the boards/rdk-s2e/ser2enet directory, the syntax of these two commands is as follow:

```
perl ../../../third_party/lwip-1.3.0/apps/httpserver_raw/makefsdata fs
fsdata-s2e.c
```

or

```
../makefsfile -i fs -o fsdata-s2e.c
```

The file system API functions are contained in `fs_s2e.c`, with `fs_s2e.h` containing the API definitions for use by applications.

49.3.2 Function Documentation

49.3.2.1 fs_close

Close an opened file designated by the handle.

Prototype:

```
void
fs_close(struct fs_file *file)
```

Parameters:

file is the pointer to the file handle to be closed.

Description:

This function will free the memory associated with the file handle, and perform any additional actions that are required for closing this handle.

Returns:

None.

49.3.2.2 fs_open

Open a file and return a handle to the file.

Prototype:

```
struct fs_file *
fs_open(char *name)
```

Parameters:

name is the pointer to the string that contains the file name.

Description:

This function will check the file name against a list of files that require “special” handling. If the file name matches this list, then the file extensions will be enabled for dynamic file content generation. Otherwise, the file name will be compared against the list of names in in the built-in flash file system. If the file is not found, a NULL handle will be returned.

Returns:

the pointer to the file handle if found, otherwise NULL.

49.3.2.3 fs_read

Read data from the opened file.

Prototype:

```
int
fs_read(struct fs_file *file,
        char *buffer,
        int count)
```

Parameters:

file is the pointer to the file handle to be read from.

buffer is the pointer to data buffer to be filled.

count is the maximum number of data bytes to be read.

Description:

This function will fill in the buffer with up to "count" bytes of data. If there is "special" processing required for dynamic content, this function will also handle that processing as needed.

Returns:

the number of data bytes read, or -1 if the end of the file has been reached.

49.4 Ring Buffer API Functions

The ring buffer module provides ring buffer management functions to support the data flow between the serial and telnet ports.

These functions are contained in `ringbuf.c`, with `ringbuf.h` containing the API definitions for use by applications.

49.5 Serial Port API Functions

Functions

- unsigned long SerialGetBaudRate (unsigned long ulPort)
- unsigned char SerialGetDataSize (unsigned long ulPort)
- unsigned char SerialGetFlowControl (unsigned long ulPort)
- unsigned char SerialGetFlowOut (unsigned long ulPort)
- unsigned char SerialGetParity (unsigned long ulPort)
- unsigned char SerialGetStopBits (unsigned long ulPort)
- void SerialGPIOAIntHandler (void)
- void SerialGPIOBIntHandler (void)
- void SerialInit (void)
- void SerialPurgeData (unsigned long ulPort, unsigned char ucPurgeCommand)
- long SerialReceive (unsigned long ulPort)
- unsigned long SerialReceiveAvailable (unsigned long ulPort)
- void SerialSend (unsigned long ulPort, unsigned char ucChar)

- tBoolean SerialSendFull (unsigned long ulPort)
- void SerialSetBaudRate (unsigned long ulPort, unsigned long ulBaudRate)
- void SerialSetCurrent (unsigned long ulPort)
- void SerialSetDataSize (unsigned long ulPort, unsigned char ucDataSize)
- void SerialSetDefault (unsigned long ulPort)
- void SerialSetFactory (unsigned long ulPort)
- void SerialSetFlowControl (unsigned long ulPort, unsigned char ucFlowControl)
- void SerialSetFlowOut (unsigned long ulPort, unsigned char ucFlowValue)
- void SerialSetParity (unsigned long ulPort, unsigned char ucParity)
- void SerialSetStopBits (unsigned long ulPort, unsigned char ucStopBits)
- void SerialUART0IntHandler (void)
- void SerialUART1IntHandler (void)

49.5.1 Detailed Description

The serial driver provides a ring buffer structure as the interface between the UART hardware and the UART client (for example, telnet session). A simple API requiring only the UART port number (for example, 0, 1) is all that is provide at this time.

These functions are contained in `serial.c`, with `serial.h` containing the API definitions for use by applications.

49.5.2 Function Documentation

49.5.2.1 SerialGetBaudRate

Queries the serial port baud rate.

Prototype:

```
unsigned long
SerialGetBaudRate(unsigned long ulPort)
```

Parameters:

ulPort is the serial port number to be accessed.

Description:

This function will read the uart configuration and return the currently configured baud rate for the selected port.

Returns:

The current baud rate of the serial port.

49.5.2.2 SerialGetDataSize

Queries the serial port data size.

Prototype:

```
unsigned char
SerialGetDataSize(unsigned long ulPort)
```

Parameters:

ulPort is the serial port number to be accessed.

Description:

This function will read the uart configuration and return the currently configured data size for the selected port.

Returns:

None.

49.5.2.3 SerialGetFlowControl

Queries the serial port flow control.

Prototype:

```
unsigned char
SerialGetFlowControl(unsigned long ulPort)
```

Parameters:

ulPort is the serial port number to be accessed.

Description:

This function will return the currently configured flow control for the selected port.

Returns:

None.

49.5.2.4 SerialGetFlowOut

Gets the serial port flow control output signal.

Prototype:

```
unsigned char
SerialGetFlowOut(unsigned long ulPort)
```

Parameters:

ulPort is the UART port number to be accessed.

Description:

This function will set the flow control output pin to a specified value.

Returns:

Returns *SERIAL_FLOW_OUT_SET* or *SERIAL_FLOW_OUT_CLEAR*.

49.5.2.5 SerialGetParity

Queries the serial port parity.

Prototype:

```
unsigned char
SerialGetParity(unsigned long ulPort)
```

Parameters:

ulPort is the serial port number to be accessed.

Description:

This function will read the uart configuration and return the currently configured parity for the selected port.

Returns:

Returns the current parity setting for the port. This will be one of /b SERIAL_PARITY_NONE, /b SERIAL_PARITY_ODD, /b SERIAL_PARITY_EVEN, /b SERIAL_PARITY_MARK, or /b SERIAL_PARITY_SPACE.

49.5.2.6 SerialGetStopBits

Queries the serial port stop bits.

Prototype:

```
unsigned char
SerialGetStopBits(unsigned long ulPort)
```

Parameters:

ulPort is the serial port number to be accessed.

Description:

This function will read the uart configuration and return the currently configured stop bits for the selected port.

Returns:

None.

49.5.2.7 SerialGPIOAIntHandler

Handles the GPIO A interrupt for flow control (port 1).

Prototype:

```
void
SerialGPIOAIntHandler(void)
```

Description:

This function is called when the GPIO port A generates an interrupt. An interrupt will be generated when the InBound flow control signal changes levels (rising/falling edge). A notification function will be called to inform the corresponding telnet session that the flow control signal has changed.

Returns:
None.

49.5.2.8 SerialGPIOBIntHandler

Handles the GPIO B interrupt for flow control (port 0).

Prototype:
```
void
SerialGPIOBIntHandler(void)
```

Description:
This function is called when the GPIO port B generates an interrupt. An interrupt will be generated when the InBound flow control signal changes levels (rising/falling edge). A notification function will be called to inform the corresponding telnet session that the flow control signal has changed.

Returns:
None.

49.5.2.9 SerialInit

Initializes the serial port driver.

Prototype:
```
void
SerialInit(void)
```

Description:
This function initializes and configures the serial port driver.

Returns:
None.

49.5.2.10 SerialPurgeData

Purges the serial port data queue(s).

Prototype:
```
void
SerialPurgeData(unsigned long ulPort,
                unsigned char ucPurgeCommand)
```

Parameters:
ulPort is the serial port number to be accessed.
ucPurgeCommand is the command indicating which queue's to purge.

Description:
This function will purge data from the tx, rx, or both serial port queues.

Returns:
None.

49.5.2.11 SerialReceive

Receives a character from the UART.

Prototype:
```
long
SerialReceive(unsigned long ulPort)
```

Parameters:
ulPort is the UART port number to be accessed.

Description:
This function sends a character to the UART. The character will either be directly written into the UART FIFO or into the UART transmit buffer, as appropriate.

Returns:
None.

49.5.2.12 SerialReceiveAvailable

Returns number of characters available in the serial ring buffer.

Prototype:
```
unsigned long
SerialReceiveAvailable(unsigned long ulPort)
```

Parameters:
ulPort is the UART port number to be accessed.

Description:
This function will return the number of characters available in the serial ring buffer.

Returns:
The number of characters available in the ring buffer..

49.5.2.13 SerialSend

Sends a character to the UART.

Prototype:
```
void
SerialSend(unsigned long ulPort,
           unsigned char ucChar)
```

Parameters:
ulPort is the UART port number to be accessed.
ucChar is the character to be sent.

Description:
This function sends a character to the UART. The character will either be directly written into the UART FIFO or into the UART transmit buffer, as appropriate.

Returns:
None.

49.5.2.14 SerialSendFull

Checks the availablity of the serial port output buffer.

Prototype:
```
tBoolean
SerialSendFull(unsigned long ulPort)
```

Parameters:
ulPort is the UART port number to be accessed.

Description:
This function checks to see if there is room on the UART transmit buffer for additional data.

Returns:
the number of bytes available in the serial transmit ring buffer.

49.5.2.15 SerialSetBaudRate

Configures the serial port baud rate.

Prototype:
```
void
SerialSetBaudRate(unsigned long ulPort,
                  unsigned long ulBaudRate)
```

Parameters:
ulPort is the serial port number to be accessed.
ulBaudRate is the new baud rate for the serial port.

Description:
This function configures the serial port's baud rate. The current configuration for the serial port will be read. The baud rate will be modified, and the port will be reconfigured.

Returns:
None.

49.5.2.16 SerialSetCurrent

Configures the serial port according to the current working parameter values.

Prototype:

```
void
SerialSetCurrent(unsigned long ulPort)
```

Parameters:

ulPort is the UART port number to be accessed. Valid values are 0 and 1.

Description:

This function configures the serial port according to the current working parameters in g_sParameters.sPort for the specified port. The actual parameter set is then read back and g_sParameters.sPort updated to ensure that the structure is correctly synchronized with the hardware.

Returns:

None.

49.5.2.17 SerialSetDataSize

Configures the serial port data size.

Prototype:

```
void
SerialSetDataSize(unsigned long ulPort,
                  unsigned char ucDataSize)
```

Parameters:

ulPort is the serial port number to be accessed.

ucDataSize is the new data size for the serial port.

Description:

This function configures the serial port's data size. The current configuration for the serial port will be read. The data size will be modified, and the port will be reconfigured.

Returns:

None.

49.5.2.18 SerialSetDefault

Configures the serial port to a default setup.

Prototype:

```
void
SerialSetDefault(unsigned long ulPort)
```

Parameters:

ulPort is the UART port number to be accessed.

Description:

This function resets the serial port to a default configuration.

Returns:

None.

49.5.2.19 SerialSetFactory

Configures the serial port to the factory default setup.

Prototype:

```
void
SerialSetFactory(unsigned long ulPort)
```

Parameters:

ulPort is the UART port number to be accessed.

Description:

This function resets the serial port to a default configuration.

Returns:

None.

49.5.2.20 SerialSetFlowControl

Configures the serial port flow control option.

Prototype:

```
void
SerialSetFlowControl(unsigned long ulPort,
                     unsigned char ucFlowControl)
```

Parameters:

ulPort is the UART port number to be accessed.

ucFlowControl is the new flow control setting for the serial port.

Description:

This function configures the serial port's flow control. This function will enable/disable the flow control interrupt and the uart transmitter based on the value of the flow control setting and/or the flow control input signal.

Returns:

None.

49.5.2.21 SerialSetFlowOut

Sets the serial port flow control output signal.

Prototype:

```
void
SerialSetFlowOut(unsigned long ulPort,
                 unsigned char ucFlowValue)
```

Parameters:

ulPort is the UART port number to be accessed.

ucFlowValue is the value to program to the flow control pin. Valid values are /b SERIAL_FLOW_OUT_SET and /b SERIAL_FLOW_OUT_CLEAR.

Description:
This function will set the flow control output pin to a specified value.

Returns:
None.

49.5.2.22 SerialSetParity

Configures the serial port parity.

Prototype:
```
void
SerialSetParity(unsigned long ulPort,
                unsigned char ucParity)
```

Parameters:
ulPort is the serial port number to be accessed.
ucParity is the new parity for the serial port.

Description:
This function configures the serial port's parity. The current configuration for the serial port will be read. The parity will be modified, and the port will be reconfigured.

Returns:
None.

49.5.2.23 SerialSetStopBits

Configures the serial port stop bits.

Prototype:
```
void
SerialSetStopBits(unsigned long ulPort,
                  unsigned char ucStopBits)
```

Parameters:
ulPort is the serial port number to be accessed.
ucStopBits is the new stop bits for the serial port.

Description:
This function configures the serial port's stop bits. The current configuration for the serial port will be read. The stop bits will be modified, and the port will be reconfigured.

Returns:
None.

49.5.2.24 SerialUART0IntHandler

Handles the UART0 interrupt.

Prototype:

```
void
SerialUART0IntHandler(void)
```

Description:

This function is called when the UART generates an interrupt. An interrupt will be generated when data is received and when the transmit FIFO becomes half empty. The transmit and receive FIFOs are processed as appropriate.

Returns:

None.

49.5.2.25 SerialUART1IntHandler

Handles the UART1 interrupt.

Prototype:

```
void
SerialUART1IntHandler(void)
```

Description:

This function is called when the UART generates an interrupt. An interrupt will be generated when data is received and when the transmit FIFO becomes half empty. The transmit and receive FIFOs are processed as appropriate.

Returns:

None.

49.6 Telnet Port API Functions

Data Structures

- tTelnetSessionData

Defines

- OPT_FLAG_DO_SUPPRESS_GA
- OPT_FLAG_SERVER
- OPT_FLAG_WILL_SUPPRESS_GA

Enumerations

- tRFC2217State
- tTCPState
- tTelnetState

Functions

- void TelnetClose (unsigned long ulSerialPort)
- unsigned short TelnetGetLocalPort (unsigned long ulSerialPort)
- unsigned short TelnetGetRemotePort (unsigned long ulSerialPort)
- void TelnetHandler (void)
- void TelnetInit (void)
- void TelnetListen (unsigned short usTelnetPort, unsigned long ulSerialPort)
- void TelnetNotifyModemState (unsigned long ulPort, unsigned char ucModemState)
- void TelnetOpen (unsigned long ulIPAddr, unsigned short usTelnetRemotePort, unsigned short usTelnetLocalPort, unsigned long ulSerialPort)

49.6.1 Detailed Description

The telnet protocol (as defined by RFC854) is used to make the connection across the network. In its simplest form, a telnet client is simply a TCP connect to the appropriate port. Telnet interprets 0xff as a command indicator (known as the Interpret As Command, or IAC, byte). Consecutive IAC bytes are used to transfer an actual 0xff byte; thus, the only special processing required is to translate 0xff to 0xff 0xff when sending, and to translate 0xff 0xff to 0xff when receiving.

The WILL, WONT, DO, DONT option negotiation protocol is also imlpemented. This is a simple means of determining if capabilities are present, and for enabling or disabling features that do not require configuration. Through the use of this negotiation protocol, telnet clients and servers are able to easily match capabilities and avoid trying to configure features that are not shared by both ends of the connection (which would therefore result in the negotiation sequence being sent as actual data instead of being absorbed by the client or server).

In this implementation, only the SUPPRESS_GA and RFC 2217 options are supported; all other options are negatively responded to in order to prevent the client from trying to use them.

These functions are contained in `telnet.c`, with `telnet.h` containing the API definitions for use by applications.

49.6.2 Data Structure Documentation

49.6.2.1 tTelnetSessionData

Definition:

```
typedef struct
{
    tcp_pcb *pConnectPCB;
    tcp_pcb *pListenPCB;
```

```
    tTCPState eTCPState;
    tTelnetState eTelnetState;
    unsigned short usTelnetLocalPort;
    unsigned short usTelnetRemotePort;
    unsigned long ulTelnetRemoteIP;
    unsigned char ucFlags;
    unsigned long ulConnectionTimeout;
    unsigned long ulMaxTimeout;
    unsigned long ulSerialPort;
    pbuf *pBufQ[PBUF_POOL_SIZE];
    int iBufQRead;
    int iBufQWrite;
    pbuf *pBufHead;
    pbuf *pBufCurrent;
    unsigned long ulBufIndex;
    unsigned long ulLastTCPSendTime;
}
tTelnetSessionData
```

Members:

pConnectPCB This value holds the pointer to the TCP PCB associated with this connected telnet session.

pListenPCB This value holds the pointer to the TCP PCB associated with this listenting telnet session.

eTCPState The current state of the TCP session.

eTelnetState The current state of the telnet option parser.

usTelnetLocalPort The listen port for the telnet server or the local port for the telnet client.

usTelnetRemotePort The remote port that the telnet client connects to.

ulTelnetRemoteIP The remote address that the telnet client connects to.

ucFlags Flags for various options associated with the telnet session.

ulConnectionTimeout A counter for the TCP connection timeout.

ulMaxTimeout The max time for TCP connection timeout counter.

ulSerialPort This value holds the UART Port Number for this telnet session.

pBufQ This value holds an array of pbufs.

iBufQRead This value holds the read index for the pbuf queue.

iBufQWrite This value holds the write index for the pbuf queue.

pBufHead This value holds the head of the pbuf that is currently being processed (that has been popped from the queue).

pBufCurrent This value holds the actual pbuf that is being processed within the pbuf chain pointed to by the pbuf head.

ulBufIndex This value holds the offset into the payload section of the current pbuf.

ulLastTCPSendTime The amount of time passed since rx byte count has changed.

Description:

This structure is used holding the state of a given telnet session.

49.6.3 Define Documentation

49.6.3.1 OPT_FLAG_DO_SUPPRESS_GA

Definition:

```
#define OPT_FLAG_DO_SUPPRESS_GA
```

Description:

The bit in the flag that is set when the remote client has sent a DO request for SUPRESS_GA and the server has accepted it.

49.6.3.2 OPT_FLAG_SERVER

Definition:

```
#define OPT_FLAG_SERVER
```

Description:

The bit in the flag that is set when a connection is operating as a telnet server. If clear, this implies that this connection is a telnet client.

49.6.3.3 OPT_FLAG_WILL_SUPPRESS_GA

Definition:

```
#define OPT_FLAG_WILL_SUPPRESS_GA
```

Description:

The bit in the flag that is set when the remote client has sent a WILL request for SUPRESS_GA and the server has accepted it.

49.6.4 Enumeration Documentation

49.6.4.1 tRFC2217State

Description:

The possible states of the telnet COM-PORT option parser.

Enumerators:

STATE_2217_GET_COMMAND The telnet COM-PORT option parser is ready to process the first byte of data, which is the sub-option to be processed.

STATE_2217_GET_DATA The telnet COM-PORT option parser is processing data bytes for the specified command/sub-option.

STATE_2217_GET_DATA_IAC The telnet COM-PORT option parser has received an IAC in the data stream.

49.6.4.2 tTCPState

Description:
The possible states of the TCP session.

Enumerators:
STATE_TCP_IDLE The TCP session is idle. No connection has been attempted, nor has it been configured to listen on any port.
STATE_TCP_LISTEN The TCP session is listening (server mode).
STATE_TCP_CONNECTING The TCP session is connecting (client mode).
STATE_TCP_CONNECTED The TCP session is connected.

49.6.4.3 tTelnetState

Description:
The possible states of the telnet option parser.

Enumerators:
STATE_NORMAL The telnet option parser is in its normal mode. Characters are passed as is until an IAC byte is received.
STATE_IAC The previous character received by the telnet option parser was an IAC byte.
STATE_WILL The previous character sequence received by the telnet option parser was IAC WILL.
STATE_WONT The previous character sequence received by the telnet option parser was IAC WONT.
STATE_DO The previous character sequence received by the telnet option parser was IAC DO.
STATE_DONT The previous character sequence received by the telnet option parser was IAC DONT.
STATE_SB The previous character sequence received by the telnet option parser was IAC SB.
STATE_SB_IGNORE The previous character sequence received by the telnet option parser was IAC SB n, where n is an unsupported option.
STATE_SB_IGNORE_IAC The previous character sequence received by the telnet option parser was IAC SB n, where n is an unsupported option.
STATE_SB_RFC2217 The previous character sequence received by the telnet option parser was IAC SB COM-PORT-OPTION (in other words, RFC 2217).

49.6.5 Function Documentation

49.6.5.1 TelnetClose

Closes an existing Ethernet connection.

Prototype:
```
void
TelnetClose(unsigned long ulSerialPort)
```

Parameters:
ulSerialPort is the serial port associated with this telnet session.

Description:
This function is called when the the Telnet/TCP session associated with the specified serial port is to be closed.

Returns:
None.

49.6.5.2 TelnetGetLocalPort

Gets the current local port for a connection's telnet session.

Prototype:
```
unsigned short
TelnetGetLocalPort(unsigned long ulSerialPort)
```

Parameters:
ulSerialPort is the serial port associated with this telnet session.

Description:
This function returns the local port in use by the telnet session associated with the given serial port. If operating as a telnet server, this port is the port that is listening for an incoming connection. If operating as a telnet client, this is the local port used to connect to the remote server.

Returns:
None.

49.6.5.3 TelnetGetRemotePort

Gets the current remote port for a connection's telnet session.

Prototype:
```
unsigned short
TelnetGetRemotePort(unsigned long ulSerialPort)
```

Parameters:
ulSerialPort is the serial port associated with this telnet session.

Description:
This function returns the remote port in use by the telnet session associated with the given serial port. If operating as a telnet server, this function will return 0. If operating as a telnet client, this is the server port that the connection is using.

Returns:
None.

49.6.5.4 TelnetHandler

Handles periodic task for telnet sessions.

Prototype:

```
void
TelnetHandler(void)
```

Description:

This function is called periodically from the lwIP timer thread context. This function will handle transferring data between the UART and the the telnet sockets. The time period for this should be tuned to the UART ring buffer sizes to maintain optimal throughput.

Returns:

None.

49.6.5.5 TelnetInit

Intializes the telnet session(s) for the Serial to Ethernet Module.

Prototype:

```
void
TelnetInit(void)
```

Description:

This function initializes the telnet session data parameter block.

Returns:

None.

49.6.5.6 TelnetListen

Opens a telnet server session (listen).

Prototype:

```
void
TelnetListen(unsigned short usTelnetPort,
             unsigned long ulSerialPort)
```

Parameters:

usTelnetPort is the telnet port number to listen on.

ulSerialPort is the serial port associated with this telnet session.

Description:

This function establishes a TCP session in listen mode as a telnet server.

Returns:

None.

49.6.5.7 TelnetNotifyModemState

Handles RFC2217 modem state notification.

Prototype:

```
void
TelnetNotifyModemState(unsigned long ulPort,
                       unsigned char ucModemState)
```

Parameters:

ulPort is the serial port for which the modem state changed.

ucModemState is the new modem state.

Description:

This function should be called by the serial port code when the modem state changes. If RFC2217 is enabled, a notification message will be sent.

Returns:

None.

49.6.5.8 TelnetOpen

Opens a telnet server session (client).

Prototype:

```
void
TelnetOpen(unsigned long ulIPAddr,
           unsigned short usTelnetRemotePort,
           unsigned short usTelnetLocalPort,
           unsigned long ulSerialPort)
```

Parameters:

ulIPAddr is the IP address of the telnet server.

usTelnetRemotePort is port number of the telnet server.

usTelnetLocalPort is local port number to connect from.

ulSerialPort is the serial port associated with this telnet session.

Description:

This function establishes a TCP session by attempting a connection to a telnet server.

Returns:

None.

49.7 Universal Plug and Play API Functions

Functions

- void UPnPHandler (unsigned long ulTimeMS)
- void UPnPInit (void)

- void UPnPStart (void)
- void UPnPStop (void)

49.7.1 Detailed Description

The UPnP Module provides the functions necessary to support UPnP on the network interface.

These functions are contained in `upnp.c`, with `upnp.h` containing the API definitions for use by applications.

49.7.2 Function Documentation

49.7.2.1 UPnPHandler

Handles Ethernet interrupt for UPnP sessions.

Prototype:

```
void
UPnPHandler(unsigned long ulTimeMS)
```

Parameters:

ulTimeMS is the absolute time (as maintained by the lwip handler) in ms.

Description:

This function should be called on a regular periodic basis to handle the various timers and process any buffers for the UPnP sessions.

Returns:

None.

49.7.2.2 UPnPInit

Intializes the UPnP session for the Serial to Ethernet Module.

Prototype:

```
void
UPnPInit(void)
```

Description:

This function initializes and configures the UPnP session for the module.

Returns:

None.

49.7.2.3 UPnPStart

Starts listening for UPnP requests.

Prototype:

```
void
UPnPStart(void)
```

Description:
This function sets up the two ports which listen for UPnP location and discovery requests.

Returns:
None.

49.7.2.4 UPnPStop

Broadcasts a byebye message and stop UPnP discovery.

Prototype:

```
void
UPnPStop(void)
```

Description:
This function broadcasts an SSDP byebye message indicating that the UPnP device is no longer available then frees resources associated with UPnP discovery and location.

Returns:
None.

49.8 Examples

All of these examples reside in the `boards/rdk-s2e` subdirectory of the peripheral driver library source distribution.

Boot Loader (boot_eth)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses Ethernet to load an application.

Serial To Ethernet Module (ser2enet)

The Serial to Ethernet Converter provides a means of accessing the UART on a Stellaris device via a network connection. The UART can be connected to the UART on a non-networked device, providing the ability to access the device via a network. This can be useful to overcome the cable length limitations of a UART connection (in fact, the cable can become thousands of miles long) and to provide networking capability to existing devices without modifing the device's operation.

The converter can be configured to use a static IP configuration or to use DHCP to obtain its IP configuration. Since the converter is providing a telnet server, the effective use of DHCP requires a reservation in the DHCP server so that the converter gets the same IP address each time it is connected to the network.

Company Information

Founded in 2004, Luminary Micro, Inc. designs, markets, and sells ARM Cortex-M3-based microcontrollers (MCUs). Austin, Texas-based Luminary Micro is the lead partner for the Cortex-M3 processor, delivering the world's first silicon implementation of the Cortex-M3 processor. Luminary Micro's introduction of the Stellaris family of products provides 32-bit performance for the same price as current 8- and 16-bit microcontroller designs. With entry-level pricing at $1.00 for an ARM technology-based MCU, Luminary Micro's Stellaris product line allows for standardization that eliminates future architectural upgrades or software tool changes.

Luminary Micro, Inc.
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com
sales@luminarymicro.com

Support Information

For support on Luminary Micro products, contact:

support@luminarymicro.com
+1-512-279-8800, ext 3

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265