




# **TivaWare™ Utilities Library**

## **USER'S GUIDE**

---

# Copyright

Copyright © 2013 Texas Instruments Incorporated. All rights reserved. Tiva and TivaWare are trademarks of Texas Instruments Instruments. ARM and Thumb are registered trademarks and Cortex is a trademark of ARM Limited. Other names and brands may be claimed as the property of others.

 Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this document.

Texas Instruments  
108 Wild Basin, Suite 350  
Austin, TX 78746  
[www.ti.com/tiva-c](http://www.ti.com/tiva-c)



## Revision Information

This is version 2.0 of this document, last updated on August 29, 2013.

# Table of Contents

<b>Copyright</b>	<b>2</b>
<b>Revision Information</b>	<b>2</b>
<b>1 Introduction</b>	<b>5</b>
<b>2 Command Line Processing Module</b>	<b>7</b>
2.1 Introduction	7
2.2 API Functions	7
2.3 Programming Example	10
<b>3 CPU Usage Module</b>	<b>13</b>
3.1 Introduction	13
3.2 API Functions	13
3.3 Programming Example	14
<b>4 Flash Parameter Block Module</b>	<b>17</b>
4.1 Introduction	17
4.2 API Functions	17
4.3 Programming Example	19
<b>5 File System Wrapper Module</b>	<b>21</b>
5.1 Introduction	21
5.2 API Functions	21
5.3 Programming Example	24
<b>6 Integer Square Root Module</b>	<b>27</b>
6.1 Introduction	27
6.2 API Functions	27
6.3 Programming Example	28
<b>7 Ethernet Board Locator Module</b>	<b>29</b>
7.1 Introduction	29
7.2 API Functions	29
7.3 Programming Example	32
<b>8 lwIP Wrapper Module</b>	<b>33</b>
8.1 Introduction	33
8.2 API Functions	33
8.3 Programming Example	37
<b>9 PTPd Wrapper Module</b>	<b>39</b>
9.1 Introduction	39
9.2 API Functions	39
9.3 Programming Example	39
<b>10 Random Number Utility Module</b>	<b>41</b>
10.1 Introduction	41
10.2 API Functions	41
10.3 Programming Example	42
<b>11 Ring Buffer Module</b>	<b>45</b>
11.1 Introduction	45
11.2 API Functions	45
11.3 Programming Example	51
<b>12 Simple Task Scheduler Module</b>	<b>53</b>
12.1 Introduction	53

12.2	API Functions	53
12.3	Programming Example	58
<b>13</b>	<b>Sine Calculation Module</b>	<b>61</b>
13.1	Introduction	61
13.2	API Functions	61
13.3	Programming Example	62
<b>14</b>	<b>SMBus Stack</b>	<b>63</b>
14.1	Introduction	63
14.2	API Functions	64
14.3	Programming Example	92
<b>15</b>	<b>Software I2C Module</b>	<b>97</b>
15.1	Introduction	97
15.2	API Functions	98
15.3	Programming Example	105
<b>16</b>	<b>Software SSI Module</b>	<b>109</b>
16.1	Introduction	109
16.2	API Functions	110
16.3	Programming Example	122
<b>17</b>	<b>Software UART Module</b>	<b>127</b>
17.1	Introduction	127
17.2	API Functions	128
17.3	Programming Example	144
<b>18</b>	<b>SPI Flash Module</b>	<b>149</b>
18.1	Introduction	149
18.2	API Functions	149
18.3	Programming Example	163
<b>19</b>	<b>Ethernet Software Update Module</b>	<b>165</b>
19.1	Introduction	165
19.2	API Functions	165
19.3	Programming Example	167
<b>20</b>	<b>UART Standard IO Module</b>	<b>169</b>
20.1	Introduction	169
20.2	API Functions	170
20.3	Programming Example	176
<b>21</b>	<b>Micro Standard Library Module</b>	<b>177</b>
21.1	Introduction	177
21.2	API Functions	177
21.3	Programming Example	186
	<b>IMPORTANT NOTICE</b>	<b>188</b>

# 1 Introduction

The Texas Instruments® TivaWare™ Utilities Library is a set of board-independent utility functions that are helpful when creating applications for the Tiva™ family of ARM® Cortex™-M based microcontrollers. This is not a library in the truest sense as there is not a set of prebuilt objects in a library container. Instead, the files in this directory are intended to be built from source as part of the target application; in many cases, there are #defines that control the features and capabilities of the utility functions that can be used to tailor the utilities to the specific needs of the application.

These utilities reside in the `utils` subdirectory of the firmware development package source distribution.



## 2 Command Line Processing Module

Introduction .....	7
API Functions .....	7
Programming Example .....	10

### 2.1 Introduction

The command line processor allows a simple command line interface to be made available in an application, for example via a UART. It takes a buffer containing a string (which must be obtained by the application) and breaks it up into a command and arguments (in traditional C “argc, argv” format). The command is then found in a command table and the corresponding function in the table is called to process the command.

This module is contained in `utils/cmdline.c`, with `utils/cmdline.h` containing the API declarations for use by applications.

### 2.2 API Functions

#### Data Structures

- `tCmdLineEntry`

#### Defines

- `CMDLINE_BAD_CMD`
- `CMDLINE_INVALID_ARG`
- `CMDLINE_TOO_FEW_ARGS`
- `CMDLINE_TOO_MANY_ARGS`

#### Functions

- `int CmdLineProcess (char *pcCmdLine)`

#### Variables

- `tCmdLineEntry g_psCmdTable[]`

## 2.2.1 Data Structure Documentation

### 2.2.1.1 tCmdLineEntry

**Definition:**

```
typedef struct
{
    const char *pcCmd;
    pfnCmdLine pfnCmd;
    const char *pcHelp;
}
tCmdLineEntry
```

**Members:**

**pcCmd** A pointer to a string containing the name of the command.

**pfnCmd** A function pointer to the implementation of the command.

**pcHelp** A pointer to a string of brief help text for the command.

**Description:**

Structure for an entry in the command list table.

## 2.2.2 Define Documentation

### 2.2.2.1 CMDLINE\_BAD\_CMD

**Definition:**

```
#define CMDLINE_BAD_CMD
```

**Description:**

Defines the value that is returned if the command is not found.

### 2.2.2.2 CMDLINE\_INVALID\_ARG

**Definition:**

```
#define CMDLINE_INVALID_ARG
```

**Description:**

Defines the value that is returned if an argument is invalid.

### 2.2.2.3 CMDLINE\_TOO\_FEW\_ARGS

**Definition:**

```
#define CMDLINE_TOO_FEW_ARGS
```

**Description:**

Defines the value that is returned if there are too few arguments.



### 2.2.2.4 CMDLINE\_TOO\_MANY\_ARGS

**Definition:**

```
#define CMDLINE_TOO_MANY_ARGS
```

**Description:**

Defines the value that is returned if there are too many arguments.

## 2.2.3 Function Documentation

### 2.2.3.1 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_psCmdTable` containing `tCmdLineEntry` structures which must be provided by the application. The array must be terminated with an entry whose **pcCmd** field contains a NULL pointer.

**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.

## 2.2.4 Variable Documentation

### 2.2.4.1 g\_psCmdTable

**Definition:**

```
tCmdLineEntry g_psCmdTable[ ]
```

**Description:**

This is the command table that must be provided by the application. The last element of the array must be a structure whose `pcCmd` field contains a NULL pointer.

## 2.3 Programming Example

The following example shows how to process a command line.

```
//
// Code for the "foo" command.
//
int
ProcessFoo(int argc, char *argv[])
{
    //
    // Do something, using argc and argv if the command takes arguments.
    //
}

//
// Code for the "bar" command.
//
int
ProcessBar(int argc, char *argv[])
{
    //
    // Do something, using argc and argv if the command takes arguments.
    //
}

//
// Code for the "help" command.
//
int
ProcessHelp(int argc, char *argv[])
{
    //
    // Provide help.
    //
}

//
// The table of commands supported by this application.
//
tCmdLineEntry g_psCmdTable[] =
{
    { "foo", ProcessFoo, "The first command." },
    { "bar", ProcessBar, "The second command." },
    { "help", ProcessHelp, "Application help." },
    { 0, 0, 0 }
};

//
// Read a process a command.
//
int
Test(void)
{
    unsigned char pucCmd[256];

    //
    // Retrieve a command from the user into pucCmd.
    //
    ...

    //
    // Process the command line.
    //
}
```

```
        return (CmdLineProcess (pucCmd) );  
    }
```



## 3 CPU Usage Module

Introduction .....	13
API Functions .....	13
Programming Example .....	14

### 3.1 Introduction

The CPU utilization module uses one of the system timers and peripheral clock gating to determine the percentage of the time that the processor is being clocked. For the most part, the processor is executing code whenever it is being clocked (exceptions occur when the clocking is being configured, which only happens at startup, and when entering/exiting an interrupt handler, when the processor is performing stacking operations on behalf of the application).

The specified timer is configured to run when the processor is in run mode and to not run when the processor is in sleep mode. Therefore, the timer will only count when the processor is being clocked. Comparing the number of clocks the timer counted during a fixed period to the number of clocks in the fixed period provides the percentage utilization.

In order for this to be effective, the application must put the processor to sleep when it has no work to do (instead of busy waiting). If the processor never goes to sleep (either because of a continual stream of work to do or a busy loop), the processor utilization will be reported as 100%.

Since deep-sleep mode changes the clocking of the system, the computed processor usage may be incorrect if deep-sleep mode is utilized. The number of clocks the processor spends in run mode will be properly counted, but the timing period may not be accurate (unless extraordinary measures are taken to ensure timing period accuracy).

The accuracy of the computed CPU utilization depends upon the regularity with which [CPUUsageTick\(\)](#) is called by the application. If the CPU usage is constant, but [CPUUsageTick\(\)](#) is called sporadically, the reported CPU usage will fluctuate as well despite the fact that the CPU usage is actually constant.

This module is contained in `utils/cpu_usage.c`, with `utils/cpu_usage.h` containing the API declarations for use by applications.

### 3.2 API Functions

#### Functions

- void [CPUUsageInit](#) (uint32\_t ui32ClockRate, uint32\_t ui32Rate, uint32\_t ui32Timer)
- uint32\_t [CPUUsageTick](#) (void)

## 3.2.1 Function Documentation

### 3.2.1.1 CPUUsageInit

Initializes the CPU usage measurement module.

**Prototype:**

```
void
CPUUsageInit (uint32_t ui32ClockRate,
              uint32_t ui32Rate,
              uint32_t ui32Timer)
```

**Parameters:**

***ui32ClockRate*** is the rate of the clock supplied to the timer module.

***ui32Rate*** is the number of times per second that [CPUUsageTick\(\)](#) is called.

***ui32Timer*** is the index of the timer module to use.

**Description:**

This function prepares the CPU usage measurement module for measuring the CPU usage of the application.

**Returns:**

None.

### 3.2.1.2 CPUUsageTick

Updates the CPU usage for the new timing period.

**Prototype:**

```
uint32_t
CPUUsageTick (void)
```

**Description:**

This function, when called at the end of a timing period, will update the CPU usage.

**Returns:**

Returns the CPU usage percentage as a 16.16 fixed-point value.

## 3.3 Programming Example

The following example shows how to use the CPU usage module to measure the CPU usage where the foreground simply burns some cycles.

```
//
// The CPU usage for the most recent time period.
//
unsigned long g_ulCPUUsage;

//
// Handles the SysTick interrupt.
```

```
//
void
SysTickIntHandler(void)
{
    //
    // Compute the CPU usage for the last time period.
    //
    g_ulCPUUsage = CPUUsageTick();
}

//
// The main application.
//
int
main(void)
{
    //
    // Initialize the CPU usage module, using timer 0.
    //
    CPUUsageInit(8000000, 100, 0);

    //
    // Initialize SysTick to interrupt at 100 Hz.
    //
    SysTickPeriodSet(8000000 / 100);
    SysTickIntEnable();
    SysTickEnable();

    //
    // Loop forever.
    //
    while(1)
    {
        //
        // Delay for a little bit so that CPU usage is not zero.
        //
        SysCtlDelay(100);

        //
        // Put the processor to sleep.
        //
        SysCtlSleep();
    }
}
```





## 4 Flash Parameter Block Module

Introduction .....	17
API Functions .....	17
Programming Example .....	19

### 4.1 Introduction

The flash parameter block module provides a simple, fault-tolerant, persistent storage mechanism for storing parameter information for an application.

The [FlashPBlockInit\(\)](#) 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 erase blocks of flash to ensure fault tolerance, and the size of the parameter block must be an integral divisor of the size of an erase block. [FlashPBlockGet\(\)](#) and [FlashPBlockSave\(\)](#) are used to read and write parameter block data into the parameter region. The only constraints 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.

This module is contained in `utils/flash_pb.c`, with `utils/flash_pb.h` containing the API declarations for use by applications.

### 4.2 API Functions

#### Functions

- `uint8_t * FlashPBlockGet (void)`
- `void FlashPBlockInit (uint32_t ui32Start, uint32_t ui32End, uint32_t ui32Size)`
- `void FlashPBlockSave (uint8_t *pui8Buffer)`

#### 4.2.1 Function Documentation

##### 4.2.1.1 FlashPBlockGet

Gets the address of the most recent parameter block.

**Prototype:**

```
uint8_t *  
FlashPBlockGet (void)
```

**Description:**

This function returns the address of the most recent parameter block that is stored in flash.

**Returns:**

Returns the address of the most recent parameter block, or NULL if there are no valid parameter blocks in flash.

#### 4.2.1.2 FlashPBInit

Initializes the flash parameter block.

**Prototype:**

```
void  
FlashPBInit (uint32_t ui32Start,  
             uint32_t ui32End,  
             uint32_t ui32Size)
```

**Parameters:**

***ui32Start*** 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.

***ui32End*** 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.

***ui32Size*** 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 *ui32Start* and *ui32End*) 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 (*ui32Size*). 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, three conditions must be met. The first is *ui32Start* and *ui32End* 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 (*ui32Size*) 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. The final condition is that the size of the flash dedicated to parameter blocks (*ui32End - ui32Start*) divided by the parameter block size (*ui32Size*) must be less than or equal to 128. If not, it will not be possible in all cases to determine which parameter block is the most recent (specifically when dealing with the sequence number wrapping back to zero).

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.

**Returns:**

None.

### 4.2.1.3 FlashPBSave

Writes a new parameter block to flash.

**Prototype:**

```
void  
FlashPBSave(uint8_t *pui8Buffer)
```

**Parameters:**

***pui8Buffer*** 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.

**Returns:**

None.

## 4.3 Programming Example

The following example shows how to use the flash parameter block module to read the contents of a flash parameter block.

```
unsigned char pucBuffer[16], *pucPB;  
  
//  
// Initialize the flash parameter block module, using the last two pages of  
// a 64 KB device as the parameter block.  
//  
FlashPBInit(0xf800, 0x10000, 16);  
  
//  
// Read the current parameter block.  
//  
pucPB = FlashPBGet();  
if(pucPB)  
{  
    memcpy(pucBuffer, pucPB);  
}
```



## 5 File System Wrapper Module

Introduction .....	21
API Functions .....	21
Programming Example .....	24

### 5.1 Introduction

The file system wrapper module allows several binary file system images and FatFs drives to be mounted simultaneously and referenced as if part of a single file system with each mount point identified by name. This is useful in applications which make use of SDCard and USB Mass Storage Class devices, allowing these to be referenced as, for example "/sdcard" and "/usb" respectively.

Mount points are defined using an array of structures, each entry of which describes a single mount and provides its name and details of the actual file system that is to be used to support that mount.

This module is contained in `utils/fswrapper.c`, with `utils/fswrapper.h` containing the API declarations for use by applications.

### 5.2 API Functions

#### Functions

- void [fs\\_close](#) (struct fs\_file \*phFile)
- bool [fs\\_init](#) (fs\_mount\_data \*psMountPoints, uint32\_t ui32NumMountPoints)
- bool [fs\\_map\\_path](#) (const char \*pcPath, char \*pcMapped, int iLen)
- fs\_file \* [fs\\_open](#) (const char \*pcName)
- int [fs\\_read](#) (struct fs\_file \*phFile, char \*pcBuffer, int iCount)
- void [fs\\_tick](#) (uint32\_t ui32TickMS)

#### 5.2.1 Function Documentation

##### 5.2.1.1 fs\_close

Closes a file.

#### Prototype:

```
void
fs_close(struct fs_file *phFile)
```

#### Parameters:

**phFile** is the handle of the file that is to be closed. This will have been returned by an earlier call to [fs\\_open\(\)](#).

#### Description:

This function closes the file identified by *phFile* and frees all resources associated with the file handle.

**Returns:**

None.

### 5.2.1.2 fs\_init

Initializes the file system wrapper.

**Prototype:**

```
bool
fs_init(fs_mount_data *psMountPoints,
        uint32_t ui32NumMountPoints)
```

**Parameters:**

**psMountPoints** points to an array of fs\_mount\_data structures. Each element in the array maps a top level directory name to a particular file system image or to the FAT file system and a logical drive number.

**ui32NumMountPoints** provides the number of populated elements in the *psMountPoints* array.

**Description:**

This function should be called to initialize the file system wrapper and provide it with the information required to access the files in multiple file system images via a single filename space.

Each entry in *psMountPoints* describes a top level directory in the unified namespace and indicates to fswrapper where the files for that directory can be found. Each entry can describe either a file system image in system memory or a logical disk handled via the FatFs file system driver.

For example, consider the following 3 entry mount point table:

```
{
    { "internal",  &g_pui8FSImage,    0,  NULL,        NULL },
    { "sdcard",    NULL,              0,  SDCardEnable, SDCardDisable },
    { NULL,        &g_pui8FSDefault,  0,  NULL,        NULL }
}
```

Requests to open file “/internal/index.html” will be handled by attempting to open “/index.html” in the internal file system pointed to by *g\_pui8FSImage*. Similarly, opening “/sdcard/images/logo.gif” will result in a call to the FAT *f\_open* function requesting “0:/images/logo.gif”. If a request to open “index.htm” is received, this is handled by attempting to open “index.htm” in the default internal file system image, *g\_pui8FSDefault*.

**Returns:**

Returns **true** on success or **false** on failure.

### 5.2.1.3 fs\_map\_path

Maps a path string containing mount point names to a path suitable for use in calls to the FatFs APIs.

**Prototype:**

```
bool
fs_map_path(const char *pcPath,
```

```
char *pcMapped,  
int iLen)
```

**Parameters:**

**pcPath** points to a string containing a path in the namespace defined by the mount information passed to [fs\\_init\(\)](#).

**pcMapped** points to a buffer into which the mapped path string will be written.

**iLen** is the size, in bytes, of the buffer pointed to by pcMapped.

**Description:**

This function may be used by applications which want to make use of FatFs functions which are not directly mapped by the fswrapper layer. A path in the namespace defined by the mount points passed to function [fs\\_init\(\)](#) is translated to an equivalent path in the FatFs namespace and this may then be used in a direct call to functions such as [f\\_opendir\(\)](#) or [f\\_getfree\(\)](#).

**Returns:**

Returns **true** on success or **false** if [fs\\_init\(\)](#) has not been called, if the path provided maps to an internal file system image rather than a FatFs logical drive or if the buffer pointed to by *pcMapped* is too small to fit the output string.

#### 5.2.1.4 fs\_open

Opens a file.

**Prototype:**

```
struct fs_file *  
fs_open(const char *pcName)
```

**Parameters:**

**pcName** points to a NULL terminated string containing the path and file name to open.

**Description:**

This function opens a file and returns a handle allowing it to be read.

**Returns:**

Returns a valid file handle on success or NULL on failure.

#### 5.2.1.5 fs\_read

Reads data from an open file.

**Prototype:**

```
int  
fs_read(struct fs_file *phFile,  
char *pcBuffer,  
int iCount)
```

**Parameters:**

**phFile** is the handle of the file which is to be read. This will have been returned by a previous call to [fs\\_open\(\)](#).

**pcBuffer** points to the first byte of the buffer into which the data read from the file will be copied. This buffer must be large enough to hold *iCount* bytes.

**iCount** is the maximum number of bytes of data that are to be read from the file.

**Description:**

This function reads the next block of data from the given file into a buffer and returns the number of bytes read or -1 if the end of the file has been reached.

**Returns:**

Returns the number of bytes read from the file or -1 if the end of the file has been reached and no more data is available.

### 5.2.1.6 fs\_tick

Provides a periodic tick for the file system.

**Prototype:**

```
void  
fs_tick(uint32_t ui32TickMS)
```

**Parameters:**

**ui32TickMS** is the number of milliseconds which have elapsed since the last time this function was called.

**Description:**

Applications making use of the file system wrapper with underlying FatFs drives must call this function at least once every 10 milliseconds to provide a time reference for use by the file system. It is typically called in the context of the application's SysTick interrupt handler or from the handler of some other timer interrupt.

If only binary file system images are in use, this function need not be called.

**Returns:**

None

## 5.3 Programming Example

The following example shows how to set up the file system wrapper with two mount points, one for a flash-based file system image and the other for a FAT file system on an SDCard.

```
/** *****  
/**  
/** This array describes the various file system mount points. These are passed  
/** to the fswrapper module which allows us to use helpful paths to access the  
/** various file systems installed via a single namespace.  
/**  
/** FS_ROOT is a pointer to a binary file system image (as can be generated by  
/** the makefsfile utility) located in system flash.  
/**  
/** *****  
static fs_mount_data g_psMountData[] =  
{
```



```
    {"sdcard",    0,          0, 0, 0}, // SDCard - FAT logical drive 0
    {"usb",      0,          1, 0, 0}, // USB flash stick - FAT logical drive 1
    {NULL,       (unsigned char *)FS_ROOT, 0, 0, 0} // Default root directory
};

void AccessFile(void)
{
    struct fs_file *fhSDCard;
    struct fs_file *fhFlash;

    //
    // Initialize the various file systems we will be using.
    //
    fs_init(g_psMountData, 3);

    //
    // Open a file on the SDCard
    //
    fhSDCard = fs_open("/sdcard/index.htm");

    //
    // Open a file in the flash file system image. The default mount point
    // identified by the "NULL" name pointer in g_psMountData is used if the
    // first directory in the path does not match any other mount point in the
    // table.
    //
    fhFlash = fs_open("/images/logo.gif");

    //
    // Do something useful with the files here.
    //

    //
    // Close our files.
    //
    fs_close(fhFlash);
    fs_close(fhSDCard);
}
```



## 6 Integer Square Root Module

Introduction .....	27
API Functions .....	27
Programming Example .....	28

### 6.1 Introduction

The integer square root module provides an integer version of the square root operation that can be used instead of the floating point version provided in the C library. The algorithm used is a derivative of the manual pencil-and-paper method that used to be taught in school, and is closely related to the pencil-and-paper division method that is likely still taught in school.

For full details of the algorithm, see the article by Jack W. Crenshaw in the February 1998 issue of Embedded System Programming. It can be found online at <http://www.embedded.com/98/9802fe2.htm>.

This module is contained in `utils/isqrt.c`, with `utils/isqrt.h` containing the API declarations for use by applications.

### 6.2 API Functions

#### Functions

- `uint32_t isqrt (uint32_t ui32Value)`

#### 6.2.1 Function Documentation

##### 6.2.1.1 isqrt

Compute the integer square root of an integer.

**Prototype:**

```
uint32_t  
isqrt (uint32_t ui32Value)
```

**Parameters:**

***ui32Value*** is the value whose square root is desired.

**Description:**

This function will compute the integer square root of the given input value. Since the value returned is also an integer, it is actually better defined as the largest integer whose square is less than or equal to the input value.

**Returns:**

Returns the square root of the input value.

## 6.3 Programming Example

The following example shows how to compute the square root of a number.

```
unsigned long ulValue;  
  
//  
// Get the square root of 52378. The result returned will be 228, which is  
// the largest integer less than or equal to the square root of 52378.  
//  
ulValue = isqrt(52378);
```

## 7 Ethernet Board Locator Module

Introduction .....	29
API Functions .....	29
Programming Example .....	32

### 7.1 Introduction

The locator module offers a simple way to add Ethernet board locator capability to an application which is using the lwIP TCP/IP stack. Applications running the locator service will be detected by the `finder` application which can be found in the `tools` directory of the TivaWare installation.

APIs offered by the locator module allow an application to set various fields which are communicated to the `finder` application when it enumerates boards on the network. These fields include an application-specified name, the MAC address of the board, the board ID and the client IP address.

This module is contained in `utils/locator.c`, with `utils/locator.h` containing the API declarations for use by applications.

### 7.2 API Functions

#### Functions

- void [LocatorAppTitleSet](#) (const char \*pcAppTitle)
- void [LocatorBoardIDSet](#) (uint32\_t ui32ID)
- void [LocatorBoardTypeSet](#) (uint32\_t ui32Type)
- void [LocatorClientIPSet](#) (uint32\_t ui32IP)
- void [LocatorInit](#) (void)
- void [LocatorMACAddrSet](#) (uint8\_t \*pui8MACArray)
- void [LocatorVersionSet](#) (uint32\_t ui32Version)

#### 7.2.1 Function Documentation

##### 7.2.1.1 LocatorAppTitleSet

Sets the application title in the locator response packet.

#### Prototype:

```
void
LocatorAppTitleSet(const char *pcAppTitle)
```

#### Parameters:

***pcAppTitle*** is a pointer to the application title string.

**Description:**

This function sets the application title in the locator response packet. The string is truncated at 64 characters if it is longer (without a terminating 0), and is zero-filled to 64 characters if it is shorter.

**Returns:**

None.

### 7.2.1.2 LocatorBoardIDSet

Sets the board ID in the locator response packet.

**Prototype:**

```
void  
LocatorBoardIDSet (uint32_t ui32ID)
```

**Parameters:**

***ui32ID*** is the ID of the board.

**Description:**

This function sets the board ID field in the locator response packet.

**Returns:**

None.

### 7.2.1.3 LocatorBoardTypeSet

Sets the board type in the locator response packet.

**Prototype:**

```
void  
LocatorBoardTypeSet (uint32_t ui32Type)
```

**Parameters:**

***ui32Type*** is the type of the board.

**Description:**

This function sets the board type field in the locator response packet.

**Returns:**

None.

### 7.2.1.4 LocatorClientIPSet

Sets the client IP address in the locator response packet.

**Prototype:**

```
void  
LocatorClientIPSet (uint32_t ui32IP)
```

**Parameters:**

***ui32IP*** is the IP address of the currently connected client.

**Description:**

This function sets the IP address of the currently connected client in the locator response packet. The IP should be set to 0.0.0.0 if there is no client connected. It should never be set for devices that do not have a strict one-to-one mapping of client to server (for example, a web server).

**Returns:**

None.

### 7.2.1.5 LocatorInit

Initializes the locator service.

**Prototype:**

```
void  
LocatorInit (void)
```

**Description:**

This function prepares the locator service to handle device discovery requests. A UDP server is created and the locator response data is initialized to all empty.

**Returns:**

None.

### 7.2.1.6 LocatorMACAddrSet

Sets the MAC address in the locator response packet.

**Prototype:**

```
void  
LocatorMACAddrSet (uint8_t *pui8MACArray)
```

**Parameters:**

***pui8MACArray*** is the MAC address of the network interface.

**Description:**

This function sets the MAC address of the network interface in the locator response packet.

**Returns:**

None.

### 7.2.1.7 LocatorVersionSet

Sets the firmware version in the locator response packet.

**Prototype:**

```
void  
LocatorVersionSet (uint32_t ui32Version)
```

**Parameters:**

***ui32Version*** is the version number of the device firmware.

**Description:**

This function sets the version number of the device firmware in the locator response packet.

**Returns:**

None.

## 7.3 Programming Example

The following example shows how to set up the board locator service in an application which uses Ethernet and the lwIP TCP/IP stack.

```
//  
// Initialize the lwIP TCP/IP stack.  
//  
lwIPInit (pucMACAddr, 0, 0, 0, IPADDR_USE_DHCP);  
  
//  
// Setup the device locator service.  
//  
LocatorInit();  
LocatorMACAddrSet (pucMACAddr);  
LocatorAppTitleSet ("Your application name");
```



## 8 lwIP Wrapper Module

Introduction .....	33
API Functions .....	33
Programming Example .....	37

### 8.1 Introduction

The lwIP wrapper module provides a simple abstraction layer for the lwIP version 1.3.2 TCP/IP stack. The configuration of the TCP/IP stack is based on the options defined in the `lwipopts.h` file provided by the application.

The `lwIPInit()` function is used to initialize the lwIP TCP/IP stack. 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.

This module is contained in `utils/lwiplib.c`, with `utils/lwiplib.h` containing the API declarations for use by applications.

### 8.2 API Functions

#### Functions

- void `lwIPEthernetIntHandler` (void)
- void `lwIPInit` (uint32\_t ui32SysClkHz, const uint8\_t \*pui8MAC, uint32\_t ui32IPAddr, uint32\_t ui32NetMask, uint32\_t ui32GWAddr, uint32\_t ui32IPMode)
- uint32\_t `lwIPLocalGWAddrGet` (void)
- uint32\_t `lwIPLocalIPAddrGet` (void)
- void `lwIPLocalMACGet` (uint8\_t \*pui8MAC)
- uint32\_t `lwIPLocalNetMaskGet` (void)
- void `lwIPNetworkConfigChange` (uint32\_t ui32IPAddr, uint32\_t ui32NetMask, uint32\_t ui32GWAddr, uint32\_t ui32IPMode)
- void `lwIPTimerCallbackRegister` (tHardwareTimerHandler pfnTimerFunc)

#### 8.2.1 Function Documentation

##### 8.2.1.1 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.

### 8.2.1.2 lwIPInit

Initializes the lwIP TCP/IP stack.

**Prototype:**

```
void  
lwIPInit (uint32_t ui32SysClkHz,  
          const uint8_t *pui8MAC,  
          uint32_t ui32IPAddr,  
          uint32_t ui32NetMask,  
          uint32_t ui32GWAddr,  
          uint32_t ui32IPMode)
```

**Parameters:**

**ui32SysClkHz** is the current system clock rate in Hz.

**pui8MAC** is a pointer to a six byte array containing the MAC address to be used for the interface.

**ui32IPAddr** is the IP address to be used (static).

**ui32NetMask** is the network mask to be used (static).

**ui32GWAddr** is the Gateway address to be used (static).

**ui32IPMode** is the IP Address Mode. **IPADDR\_USE\_STATIC** will force static IP addressing to be used, **IPADDR\_USE\_DHCP** will force DHCP with fallback to Link Local (Auto IP), while **IPADDR\_USE\_AUTOIP** will force Link Local only.

**Description:**

This function performs initialization of the lwIP TCP/IP stack for the Ethernet MAC, including DHCP and/or AutoIP, as configured.

**Returns:**

None.

### 8.2.1.3 lwIPLocalGWAddrGet

Returns the gateway address for this interface.

**Prototype:**

```
uint32_t  
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.

#### 8.2.1.4 lwIPLocalIPAddrGet

Returns the IP address for this interface.

**Prototype:**

```
uint32_t  
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.

#### 8.2.1.5 lwIPLocalMACGet

Returns the local MAC/HW address for this interface.

**Prototype:**

```
void  
lwIPLocalMACGet(uint8_t *pui8MAC)
```

**Parameters:**

***pui8MAC*** 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 passed in *pui8MAC*.

**Returns:**

None.

#### 8.2.1.6 lwIPLocalNetMaskGet

Returns the network mask for this interface.

**Prototype:**

```
uint32_t  
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.

### 8.2.1.7 lwIPNetworkConfigChange

Change the configuration of the lwIP network interface.

**Prototype:**

```
void  
lwIPNetworkConfigChange (uint32_t ui32IPAddr,  
                          uint32_t ui32NetMask,  
                          uint32_t ui32GWAddr,  
                          uint32_t ui32IPMode)
```

**Parameters:**

**ui32IPAddr** is the new IP address to be used (static).

**ui32NetMask** is the new network mask to be used (static).

**ui32GWAddr** is the new Gateway address to be used (static).

**ui32IPMode** is the IP Address Mode. **IPADDR\_USE\_STATIC** 0 will force static IP addressing to be used, **IPADDR\_USE\_DHCP** will force DHCP with fallback to Link Local (Auto IP), while **IPADDR\_USE\_AUTOIP** 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.

### 8.2.1.8 lwIPTimerCallbackRegister

Registers an interrupt callback function to handle the IEEE-1588 timer.

**Prototype:**

```
void  
lwIPTimerCallbackRegister (tHardwareTimerHandler pfnTimerFunc)
```

**Parameters:**

**pfnTimerFunc** points to a function which is called whenever the Ethernet MAC reports an interrupt relating to the IEEE-1588 hardware timer.

**Description:**

This function allows an application to register a handler for all interrupts generated by the IEEE-1588 hardware timer in the Ethernet MAC. To allow minimal latency timer handling, the callback function provided will be called in interrupt context, regardless of whether or not lwIP is configured to operate with an RTOS. In an RTOS environment, the callback function is responsible for ensuring that all processing it performs is compatible with the low level interrupt context it is called in.

The callback function takes two parameters. The first is the base address of the MAC reporting the timer interrupt and the second is the timer interrupt status as reported by `EMACTimestampIntStatus()`. Note that `EMACTimestampIntStatus()` causes the timer interrupt sources to be cleared so the application should not call `EMACTimestampIntStatus()` within the handler.

**Returns:**  
None.

## 8.3 Programming Example

The following example shows how to use the lwIP wrapper module to initialize the lwIP stack.

```
unsigned char pucMACArray[6];

//
// Fill in the MAC array and initialize the lwIP library using DHCP.
//
lwIPInit(pucMACArray, 0, 0, 0, IPADDR_USE_DHCP);

//
// Periodically call the lwIP timer tick. In a real application, this
// would use a timer interrupt instead of an endless loop.
//
while(1)
{
    SysCtlDelay(1000);
    lwIPTimer(1);
}
```



## 9 PTPd Wrapper Module

Introduction .....	39
API Functions .....	39
Programming Example .....	39

### 9.1 Introduction

The PTPd wrapper module provides a simple way to include the open-source PTPd library in an application. Because the PTPd library has compile-time options that may vary from one application to the next, it is not practical to provide this library in object format. By including the `ptpdlib.c` module in your application's project and/or make file, the library can be included at compile-time with a single reference.

The PTPd library provides IEEE Precision Time Protocol (1588) ported to the Tiva family of Ethernet-enabled devices. This port uses lwIP as the underlying TCP/IP stack. Refer to the `enet_ptpd` sample application for the EK-6965 and EK-8962 Evaluation Kits for additional details.

This module is contained in `utils/ptpdlib.c`, with `utils/ptpdlib.h` containing the API declarations for use by applications.

### 9.2 API Functions

### 9.3 Programming Example

```
//
// Clear out all of the run time options and protocol stack options.
//
memset(&g_sRtOpts, 0, sizeof(g_sRtOpts));
memset(&g_sPTPClock, 0, sizeof(g_sPTPClock));

//
// Initialize all PTPd Run Time and Clock Options.
// Note: This code will be specific to your application
//
...

//
// Run the protocol engine for the first time to initialize the state
// machines.
//
protocol_first(&g_sRtOpts, &g_sPTPClock);

...

//
// Main Loop
//
while(1)
{
    ...
}
```

```
    //  
    // Run the protocol engine for each pass through the main process loop.  
    //  
    protocol_loop(&g_sRtOpts, &g_sPTPClock);  
  
    ...  
}
```



# 10 Random Number Utility Module

Introduction .....	41
API Functions .....	41
Programming Example .....	42

## 10.1 Introduction

This module implements an entropy-based random number generator (RNG). The API only consists of two functions: [RandomAddEntropy\(\)](#) and [RandomSeed\(\)](#). Software uses [RandomAddEntropy\(\)](#) to add random data (entropy) to the pool used by the RNG. Entropy can originate from various sources including ADC measurements, sensor readings, time between interrupts, timing user inputs (such as keyboard or mouse), etc. Essentially anything that can create random data.

This module can be used in different ways. The simplest method is to feed the entropy pool and call [RandomSeed\(\)](#) to obtain a new random number. Assuming that the entropy pool has changed since the last call to [RandomSeed\(\)](#), a new random number is returned by [RandomSeed\(\)](#). If the entropy pool has not changed since the last call to [RandomSeed\(\)](#), the same number is returned.

Software can also use this module in combination with other RNG or pseudo-random number generator (PRNG) software. For example, software can feed the entropy pool and generate seed data that is fed into another, more sophisticated RNG or PRNG. A simple example of this is to use [RandomSeed\(\)](#) to obtain the seed used by the PRNG functions located in `utils/ustdlib.c`.

How this module is used should be dictated by the requirements of the end user application.

This module is contained in `utils/random.c`, with `utils/random.h` containing the API declarations for use by applications.

## 10.2 API Functions

### Functions

- void [RandomAddEntropy](#) (uint32\_t ui32Entropy)
- uint32\_t [RandomSeed](#) (void)

### 10.2.1 Function Documentation

#### 10.2.1.1 RandomAddEntropy

Add entropy to the pool.

#### Prototype:

```
void
RandomAddEntropy(uint32_t ui32Entropy)
```

#### Parameters:

***ui32Entropy*** is an 8-bit value that is added to the entropy pool

**Description:**

This function allows the user application code to add entropy (random data) to the pool at any time.

**Returns:**

None

### 10.2.1.2 RandomSeed

Set the random number generator seed.

**Prototype:**

```
uint32_t  
RandomSeed(void)
```

**Description:**

Seed the random number generator by running a MD4 hash on the entropy pool. Note that the entropy pool may change from beneath us, but for the purposes of generating random numbers that is not a concern. Also, the MD4 hash was broken long ago, but since it is being used to generate random numbers instead of providing security this is not a concern.

**Returns:**

New seed value.

## 10.3 Programming Example

The following example shows how to produce a random number using the SysTick timer and an ADC reading to feed the entropy pool.

```
unsigned long ulRandomNumber1, ulRandomNumber2;  
  
//  
// Add entropy to the pool and generate a new random number.  
//  
RandomAddEntropy(SysTickValueGet());  
ulRandomNumber1 = RandomSeed();  
  
//  
// Add entropy to the pool and generate a new random number.  
//  
RandomAddEntropy(ulADCValue);  
ulRandomNumber2 = RandomSeed();
```

The following example shows how to use seed data from the RNG module with another piece of RNG software. This example adds entropy from various sources.

```
unsigned long ulRandomNumber1, ulRandomNumber2;  
  
//  
// Initialize the entropy pool.  
//  
RandomAddEntropy(SysTickValueGet());  
RandomAddEntropy(ulSensorValue);
```

```
RandomAddEntropy(ulADCValue);

//
// Seed the random number generator.
//
usrand(RandomSeed());

//
// Generate random numbers.
//
ulRandomNumber1 = urand();
ulRandomNumber2 = urand();
```



# 11 Ring Buffer Module

Introduction .....	45
API Functions .....	45
Programming Example .....	51

## 11.1 Introduction

The ring buffer module provides a set of functions allowing management of a block of memory as a ring buffer. This is typically used in buffering transmit or receive data for a communication channel but has many other uses including implementing queues and FIFOs.

This module is contained in `utils/ringbuf.c`, with `utils/ringbuf.h` containing the API declarations for use by applications.

## 11.2 API Functions

### Functions

- void [RingBufAdvanceRead](#) (tRingBufObject \*psRingBuf, uint32\_t ui32NumBytes)
- void [RingBufAdvanceWrite](#) (tRingBufObject \*psRingBuf, uint32\_t ui32NumBytes)
- uint32\_t [RingBufContigFree](#) (tRingBufObject \*psRingBuf)
- uint32\_t [RingBufContigUsed](#) (tRingBufObject \*psRingBuf)
- bool [RingBufEmpty](#) (tRingBufObject \*psRingBuf)
- void [RingBufFlush](#) (tRingBufObject \*psRingBuf)
- uint32\_t [RingBufFree](#) (tRingBufObject \*psRingBuf)
- bool [RingBufFull](#) (tRingBufObject \*psRingBuf)
- void [RingBufInit](#) (tRingBufObject \*psRingBuf, uint8\_t \*pui8Buf, uint32\_t ui32Size)
- void [RingBufRead](#) (tRingBufObject \*psRingBuf, uint8\_t \*pui8Data, uint32\_t ui32Length)
- uint8\_t [RingBufReadOne](#) (tRingBufObject \*psRingBuf)
- uint32\_t [RingBufSize](#) (tRingBufObject \*psRingBuf)
- uint32\_t [RingBufUsed](#) (tRingBufObject \*psRingBuf)
- void [RingBufWrite](#) (tRingBufObject \*psRingBuf, uint8\_t \*pui8Data, uint32\_t ui32Length)
- void [RingBufWriteOne](#) (tRingBufObject \*psRingBuf, uint8\_t ui8Data)

### 11.2.1 Function Documentation

#### 11.2.1.1 RingBufAdvanceRead

Remove bytes from the ring buffer by advancing the read index.

**Prototype:**

```
void
RingBufAdvanceRead(tRingBufObject *psRingBuf,
                  uint32_t ui32NumBytes)
```

**Parameters:**

***psRingBuf*** points to the ring buffer from which bytes are to be removed.  
***ui32NumBytes*** 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 *ui32NumBytes* is larger than the number of bytes currently in the buffer, the buffer is emptied.

**Returns:**

None.

### 11.2.1.2 RingBufAdvanceWrite

Add bytes to the ring buffer by advancing the write index.

**Prototype:**

```
void  
RingBufAdvanceWrite(tRingBufObject *psRingBuf,  
                    uint32_t ui32NumBytes)
```

**Parameters:**

***psRingBuf*** points to the ring buffer to which bytes have been added.  
***ui32NumBytes*** 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 *ui32NumBytes* 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.

### 11.2.1.3 RingBufContigFree

Returns number of contiguous free bytes available in a ring buffer.

**Prototype:**

```
uint32_t  
RingBufContigFree(tRingBufObject *psRingBuf)
```

**Parameters:**

***psRingBuf*** 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.

#### 11.2.1.4 RingBufContigUsed

Returns number of contiguous bytes of data stored in ring buffer ahead of the current read pointer.

**Prototype:**

```
uint32_t  
RingBufContigUsed(tRingBufObject *psRingBuf)
```

**Parameters:**

***psRingBuf*** 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.

#### 11.2.1.5 RingBufEmpty

Determines whether the ring buffer whose pointers and size are provided is empty or not.

**Prototype:**

```
bool  
RingBufEmpty(tRingBufObject *psRingBuf)
```

**Parameters:**

***psRingBuf*** 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.

#### 11.2.1.6 RingBufFlush

Empties the ring buffer.

**Prototype:**

```
void  
RingBufFlush(tRingBufObject *psRingBuf)
```

**Parameters:**

***psRingBuf*** is the ring buffer object to empty.

**Description:**

Discards all data from the ring buffer.

**Returns:**

None.

### 11.2.1.7 RingBufFree

Returns number of bytes available in a ring buffer.

**Prototype:**

```
uint32_t  
RingBufFree (tRingBufObject *psRingBuf)
```

**Parameters:**

***psRingBuf*** 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.

### 11.2.1.8 RingBufFull

Determines whether the ring buffer whose pointers and size are provided is full or not.

**Prototype:**

```
bool  
RingBufFull (tRingBufObject *psRingBuf)
```

**Parameters:**

***psRingBuf*** 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.

### 11.2.1.9 RingBufInit

Initialize a ring buffer object.

**Prototype:**

```
void  
RingBufInit (tRingBufObject *psRingBuf,  
             uint8_t *pui8Buf,  
             uint32_t ui32Size)
```



**Parameters:**

***psRingBuf*** points to the ring buffer to be initialized.  
***pui8Buf*** points to the data buffer to be used for the ring buffer.  
***ui32Size*** is the size of the buffer in bytes.

**Description:**

This function initializes a ring buffer object, preparing it to store data.

**Returns:**

None.

### 11.2.1.10 RingBufRead

Reads data from a ring buffer.

**Prototype:**

```
void  
RingBufRead(tRingBufObject *psRingBuf,  
            uint8_t *pui8Data,  
            uint32_t ui32Length)
```

**Parameters:**

***psRingBuf*** points to the ring buffer to be read from.  
***pui8Data*** points to where the data should be stored.  
***ui32Length*** is the number of bytes to be read.

**Description:**

This function reads a sequence of bytes from a ring buffer.

**Returns:**

None.

### 11.2.1.11 RingBufReadOne

Reads a single byte of data from a ring buffer.

**Prototype:**

```
uint8_t  
RingBufReadOne(tRingBufObject *psRingBuf)
```

**Parameters:**

***psRingBuf*** 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.

#### 11.2.1.12 RingBufSize

Return size in bytes of a ring buffer.

**Prototype:**

```
uint32_t  
RingBufSize (tRingBufObject *psRingBuf)
```

**Parameters:**

***psRingBuf*** 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.

#### 11.2.1.13 RingBufUsed

Returns number of bytes stored in ring buffer.

**Prototype:**

```
uint32_t  
RingBufUsed (tRingBufObject *psRingBuf)
```

**Parameters:**

***psRingBuf*** 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.

#### 11.2.1.14 RingBufWrite

Writes data to a ring buffer.

**Prototype:**

```
void  
RingBufWrite (tRingBufObject *psRingBuf,  
              uint8_t *pui8Data,  
              uint32_t ui32Length)
```

**Parameters:**

***psRingBuf*** points to the ring buffer to be written to.

***pui8Data*** points to the data to be written.

***ui32Length*** is the number of bytes to be written.

**Description:**

This function write a sequence of bytes into a ring buffer.

**Returns:**  
None.

#### 11.2.1.15 RingBufWriteOne

Writes a single byte of data to a ring buffer.

**Prototype:**  
`void  
RingBufWriteOne(tRingBufObject *psRingBuf,  
uint8_t ui8Data)`

**Parameters:**  
***psRingBuf*** points to the ring buffer to be written to.  
***ui8Data*** is the byte to be written.

**Description:**  
This function writes a single byte of data into a ring buffer.

**Returns:**  
None.

## 11.3 Programming Example

The following example shows how to pass data through the ring buffer.

```
char pcBuffer[128], pcData[16];  
tRingBufObject sRingBuf;  
  
//  
// Initialize the ring buffer.  
//  
RingBufInit(&sRingBuf, pcBuffer, sizeof(pcBuffer));  
  
//  
// Write some data into the ring buffer.  
//  
RingBufWrite(&sRingBuf, "Hello World", 11);  
  
//  
// Read the data out of the ring buffer.  
//  
RingBufRead(&sRingBuf, pcData, 11);
```



## 12 Simple Task Scheduler Module

Introduction .....	53
API Functions .....	53
Programming Example .....	58

### 12.1 Introduction

The simple task scheduler module offers an easy way to implement applications which rely upon a group of functions being called at regular time intervals. The module makes use of an application-defined task table listing functions to be called. Each task is defined by a function pointer, a parameter that will be passed to that function, the period between consecutive calls to the function and a flag indicating whether that particular task is enabled.

The scheduler makes use of the SysTick counter and interrupt to track time and calls enabled functions when the appropriate period has elapsed since the last call to that function.

In addition to providing the task table `g_psSchedulerTable[]` to the module, the application must also define a global variable `g_ulSchedulerNumTasks` containing the number of task entries in the table. The module also requires exclusive access to the SysTick hardware and the application must hook the scheduler's SysTick interrupt handler to the appropriate interrupt vector. Although the scheduler owns SysTick, functions are provided to allow the current system time to be queried and to calculate elapsed time between two system time values or between an earlier time value and the present time.

All times passed to the scheduler or returned from it are expressed in terms of system ticks. The basic system tick rate is set by the application when it initializes the scheduler module.

This module is contained in `utils/scheduler.c`, with `utils/scheduler.h` containing the API declarations for use by applications.

### 12.2 API Functions

#### Data Structures

- `tSchedulerTask`

#### Functions

- `uint32_t SchedulerElapsedTicksCalc` (`uint32_t ui32TickStart`, `uint32_t ui32TickEnd`)
- `uint32_t SchedulerElapsedTicksGet` (`uint32_t ui32TickCount`)
- `void SchedulerInit` (`uint32_t ui32TicksPerSecond`)
- `void SchedulerRun` (`void`)
- `void SchedulerSysTickIntHandler` (`void`)
- `void SchedulerTaskDisable` (`uint32_t ui32Index`)
- `void SchedulerTaskEnable` (`uint32_t ui32Index`, `bool bRunNow`)
- `uint32_t SchedulerTickCountGet` (`void`)

## Variables

- `tSchedulerTask g_psSchedulerTable[]`
- `uint32_t g_ui32SchedulerNumTasks`

## 12.2.1 Data Structure Documentation

### 12.2.1.1 tSchedulerTask

**Definition:**

```
typedef struct
{
    void (*pfnFunction) (void *);
    void *pvParam;
    uint32_t ui32FrequencyTicks;
    uint32_t ui32LastCall;
    bool bActive;
}
tSchedulerTask
```

**Members:**

***pfnFunction*** A pointer to the function which is to be called periodically by the scheduler.

***pvParam*** The parameter which is to be passed to this function when it is called.

***ui32FrequencyTicks*** The frequency the function is to be called expressed in terms of system ticks. If this value is 0, the function will be called on every call to SchedulerRun.

***ui32LastCall*** Tick count when this function was last called. This field is updated by the scheduler.

***bActive*** A flag indicating whether or not this task is active. If true, the function will be called periodically. If false, the function is disabled and will not be called.

**Description:**

The structure defining a function which the scheduler will call periodically.

## 12.2.2 Function Documentation

### 12.2.2.1 SchedulerElapsedTicksCalc

Returns the number of ticks elapsed between two times.

**Prototype:**

```
uint32_t
SchedulerElapsedTicksCalc (uint32_t ui32TickStart,
                           uint32_t ui32TickEnd)
```

**Parameters:**

***ui32TickStart*** is the system tick count for the start of the period.

***ui32TickEnd*** is the system tick count for the end of the period.

**Description:**

This function may be called by a client to determine the number of ticks which have elapsed between provided starting and ending tick counts. The function takes into account wrapping cases where the end tick count is lower than the starting count assuming that the ending tick count always represents a later time than the starting count.

**Returns:**

The number of ticks elapsed between the provided start and end counts.

### 12.2.2.2 SchedulerElapsedTicksGet

Returns the number of ticks elapsed since the provided tick count.

**Prototype:**

```
uint32_t  
SchedulerElapsedTicksGet (uint32_t ui32TickCount)
```

**Parameters:**

***ui32TickCount*** is the tick count from which to determine the elapsed time.

**Description:**

This function may be called by a client to determine how much time has passed since a particular tick count provided in the *ui32TickCount* parameter. This function takes into account wrapping of the global tick counter and assumes that the provided tick count always represents a time in the past. The returned value will, of course, be wrong if the tick counter has wrapped more than once since the passed *ui32TickCount*. As a result, please do not use this function if you are dealing with timeouts of 497 days or longer (assuming you use a 10mS tick period).

**Returns:**

The number of ticks elapsed since the provided tick count.

### 12.2.2.3 SchedulerInit

Initializes the task scheduler.

**Prototype:**

```
void  
SchedulerInit (uint32_t ui32TicksPerSecond)
```

**Parameters:**

***ui32TicksPerSecond*** sets the basic frequency of the SysTick interrupt used by the scheduler to determine when to run the various task functions.

**Description:**

This function must be called during application startup to configure the SysTick timer. This is used by the scheduler module to determine when each of the functions provided in the `g_psSchedulerTable` array is called.

The caller is responsible for ensuring that [SchedulerSysTickIntHandler\(\)](#) has previously been installed in the SYSTICK vector in the vector table and must also ensure that interrupts are enabled at the CPU level.

Note that this call does not start the scheduler calling the configured functions. All function calls are made in the context of later calls to [SchedulerRun\(\)](#). This call merely configures the SysTick interrupt that is used by the scheduler to determine what the current system time is.

**Returns:**

None.

#### 12.2.2.4 SchedulerRun

Instructs the scheduler to update its task table and make calls to functions needing called.

**Prototype:**

```
void  
SchedulerRun(void)
```

**Description:**

This function must be called periodically by the client to allow the scheduler to make calls to any configured task functions if it is their time to be called. The call must be made at least as frequently as the most frequent task configured in the `g_psSchedulerTable` array.

Although the scheduler makes use of the SysTick interrupt, all calls to functions configured in `g_psSchedulerTable` are made in the context of [SchedulerRun\(\)](#).

**Returns:**

None.

#### 12.2.2.5 SchedulerSysTickIntHandler

Handles the SysTick interrupt on behalf of the scheduler module.

**Prototype:**

```
void  
SchedulerSysTickIntHandler(void)
```

**Description:**

Applications using the scheduler module must ensure that this function is hooked to the SysTick interrupt vector.

**Returns:**

None.

#### 12.2.2.6 SchedulerTaskDisable

Disables a task and prevents the scheduler from calling it.

**Prototype:**

```
void  
SchedulerTaskDisable(uint32_t ui32Index)
```



**Parameters:**

**ui32Index** is the index of the task which is to be disabled in the global *g\_psSchedulerTable* array.

**Description:**

This function marks one of the configured tasks as inactive and prevents [SchedulerRun\(\)](#) from calling it. The task may be reenabled by calling [SchedulerTaskEnable\(\)](#).

**Returns:**

None.

### 12.2.2.7 SchedulerTaskEnable

Enables a task and allows the scheduler to call it periodically.

**Prototype:**

```
void  
SchedulerTaskEnable(uint32_t ui32Index,  
                    bool bRunNow)
```

**Parameters:**

**ui32Index** is the index of the task which is to be enabled in the global *g\_psSchedulerTable* array.

**bRunNow** is **true** if the task is to be run on the next call to [SchedulerRun\(\)](#) or **false** if one whole period is to elapse before the task is run.

**Description:**

This function marks one of the configured tasks as enabled and causes [SchedulerRun\(\)](#) to call that task periodically. The caller may choose to have the enabled task run for the first time on the next call to [SchedulerRun\(\)](#) or to wait one full task period before making the first call.

**Returns:**

None.

### 12.2.2.8 SchedulerTickCountGet

Returns the current system time in ticks since power on.

**Prototype:**

```
uint32_t  
SchedulerTickCountGet(void)
```

**Description:**

This function may be called by a client to retrieve the current system time. The value returned is a count of ticks elapsed since the system last booted.

**Returns:**

Tick count since last boot.

## 12.2.3 Variable Documentation

### 12.2.3.1 g\_psSchedulerTable

**Definition:**

```
tSchedulerTask g_psSchedulerTable[ ]
```

**Description:**

This global table must be populated by the client and contains information on each function that the scheduler is to call.

### 12.2.3.2 g\_ui32SchedulerNumTasks

**Definition:**

```
uint32_t g_ui32SchedulerNumTasks
```

**Description:**

This global variable must be exported by the client. It must contain the number of entries in the g\_psSchedulerTable array.

## 12.3 Programming Example

The following example shows how to use the task scheduler module. This code illustrates a simple application which toggles two LEDs at different rates and updates a scrolling text string on the display.

```
//*****  
//  
// Definition of the system tick rate. This results in a tick period of 10mS.  
//  
//*****  
#define TICKS_PER_SECOND 100  
  
//*****  
//  
// Prototypes of functions which will be called by the scheduler.  
//  
//*****  
static void ScrollTextBanner(void *pvParam);  
static void ToggleLED(void *pvParam);  
  
//*****  
//  
// This table defines all the tasks that the scheduler is to run, the periods  
// between calls to those tasks, and the parameter to pass to the task.  
//  
//*****  
tSchedulerTask g_psSchedulerTable[] =  
{  
    //  
    // Scroll the text banner 1 character to the left. This function is called  
    // every 20 ticks (5 times per second).  
    //  
    { ScrollTextBanner, (void *)0, 20, 0, true},  
}
```

```

//
// Toggle LED number 0 every 50 ticks (twice per second).
//
{ ToggleLED, (void *)0, 50, 0, true},

//
// Toggle LED number 1 every 100 ticks (once per second).
//
{ ToggleLED, (void *)1, 100, 0, true},
};

//*****
//
// The number of entries in the global scheduler task table.
//
//*****
unsigned long g_ulSchedulerNumTasks = (sizeof(g_psSchedulerTable) /
                                       sizeof(tSchedulerTask));

//*****
//
// This function is called by the scheduler to toggle one of two LEDs
//
//*****
static void
ToggleLED(void *pvParam)
{
    long lState;

    ulState = GPIOPinRead(LED_GPIO_BASE
                          (pvParam ? LED1_GPIO_PIN : LED0_GPIO_PIN));
    GPIOPinWrite(LED_GPIO_BASE, (pvParam ? LED1_GPIO_PIN : LED0_GPIO_PIN),
                 ~lState);
}

//*****
//
// This function is called by the scheduler to scroll a line of text on the
// display.
//
//*****
static void
ScrollTextBanner(void *pvParam)
{
    //
    // Left as an exercise for the reader.
    //
}

//*****
//
// Application main task.
//
//*****
int
main(void)
{
    //
    // Initialize system clock and any peripherals that are to be used.
    //
    SystemInit();

    //
    // Initialize the task scheduler and configure the SysTick to interrupt
    // 100 times per second.

```

```
//
SchedulerInit(TICKS_PER_SECOND);

//
// Turn on interrupts at the CPU level.
//
IntMasterEnable();

//
// Drop into the main loop.
//
while(1)
{
    //
    // Tell the scheduler to call any periodic tasks that are due to be
    // called.
    //
    SchedulerRun();
}
}
```

## 13 Sine Calculation Module

Introduction .....	61
API Functions .....	61
Programming Example .....	62

### 13.1 Introduction

This module provides a fixed-point sine function. The input angle is a 0.32 fixed-point value that is the percentage of 360 degrees. This has two benefits; the sine function does not have to handle angles that are outside the range of 0 degrees through 360 degrees (in fact, 360 degrees can not be represented since it would wrap to 0 degrees), and the computation of the angle can be simplified since it does not have to deal with wrapping at values that are not natural for binary arithmetic (such as 360 degrees or  $2\pi$  radians).

A sine table is used to find the approximate value for a given input angle. The table contains 128 entries that range from 0 degrees through 90 degrees and the symmetry of the sine function is used to determine the value between 90 degrees and 360 degrees. The maximum error caused by this table-based approach is 0.00618, which occurs near 0 and 180 degrees.

This module is contained in `utils/sine.c`, with `utils/sine.h` containing the API declarations for use by applications.

### 13.2 API Functions

#### Defines

- `cosine(ui32Angle)`

#### Functions

- `int32_t sine (uint32_t ui32Angle)`

#### 13.2.1 Define Documentation

##### 13.2.1.1 cosine

Computes an approximation of the cosine of the input angle.

**Definition:**

```
#define cosine(ui32Angle)
```

**Parameters:**

***ui32Angle*** is an angle expressed as a 0.32 fixed-point value that is the percentage of the way around a circle.

**Description:**

This function computes the cosine for the given input angle. The angle is specified in 0.32 fixed point format, and is therefore always between 0 and 360 degrees, inclusive of 0 and exclusive of 360.

**Returns:**

Returns the cosine of the angle, in 16.16 fixed point format.

## 13.2.2 Function Documentation

### 13.2.2.1 sine

Computes an approximation of the sine of the input angle.

**Prototype:**

```
int32_t  
sine(uint32_t ui32Angle)
```

**Parameters:**

**ui32Angle** is an angle expressed as a 0.32 fixed-point value that is the percentage of the way around a circle.

**Description:**

This function computes the sine for the given input angle. The angle is specified in 0.32 fixed point format, and is therefore always between 0 and 360 degrees, inclusive of 0 and exclusive of 360.

**Returns:**

Returns the sine of the angle, in 16.16 fixed point format.

## 13.3 Programming Example

The following example shows how to produce a sine wave with 7 degrees between successive values.

```
unsigned long ulValue;  
  
//  
// Produce a sine wave with each step being 7 degrees advanced from the  
// previous.  
//  
for(ulValue = 0; ; ulValue += 0x04FA4FA4)  
{  
    //  
    // Compute the sine at this angle and do something with the result.  
    //  
    sine(ulValue);  
}
```

## 14 SMBus Stack

Introduction .....	63
API Functions .....	64
Programming Example .....	92

### 14.1 Introduction

The SMBus stack takes advantage of the SMBus extensions present on the I2C module. All standard SMBus protocols are supported in the SMBus stack, including Packet Error Checking (PEC) and Address Resolution Protocol (ARP). PEC can be enabled or disabled on a per transfer basis by using the [SMBusPECEnable\(\)](#) and [SMBusPECDisable\(\)](#) functions.

The stack uses a per instance configuration data structure to define various settings for each bus. The data structure has both public and private members, and software should take care not to modify members that it does not need to. For example, the interrupt state machine is tracked via the configuration structure and can be adversely affected by modifications made by the user application. Public members include things such as the base address of the I2C peripheral being used, transmit/receive buffer pointers, transfer sizes, etc.

User application software is responsible for doing basic configuration of each I2C peripheral being used for SMBus before attempting to do any bus transactions. For example, user code must enable the GPIO ports, configure the pins, set up the functional IO mux, and enable the clock to the I2C peripheral. Everything else, including initialization of the specific I2C peripheral and interrupts, is handled via SMBus stack calls such as [SMBusMasterInit\(\)](#), [SMBusSlaveInit\(\)](#), [SMBusMasterIntEnable\(\)](#) and [SMBusSlaveIntEnable\(\)](#). When using ARP, software can optionally define a Unique Device Identification (UDID) structure to be used by the slave during the configuration phase.

As mentioned above, the SMBus stack is based on an interrupt-driven state machine. When performing master operations, an application can choose to either poll the status of a transaction using [SMBusStatusGet\(\)](#) or look at the return code from any of the SMBusMaster functions that initiate new transactions. If the SMBus instance is busy, it will return without impacting the ongoing transfer. Slave operations can also use [SMBusStatusGet\(\)](#) to query the status of an ongoing transfer. This implementation is RTOS-friendly.

On the master side, things are very straightforward, with user code needing only a call to [SMBusMasterIntProcess\(\)](#) in the simplest case. Return codes can be tracked for events such as slave NACK or other error conditions if desired. Once the stack is configured at initialization time, the user code makes calls to the various SMBusMaster functions to initiate transfers using specific SMBus protocols.

The SMBus slave requires much more interaction from the user application. Since the slave is “dumb”, meaning that it doesn’t know which protocol to use until software tells it, the slave interrupt service routine requires much more code than the master case. The typical flow would be a call to [SMBusSlaveIntProcess\(\)](#) followed by code that analyses the return code and the first data byte received. The typical SMBus flow is to have the master send a command byte first. Once the ISR analyzes the first data byte, it must set stack-specific flags for things such as process call or block transfers so that the state machine functions correctly.

This module is contained in `utils/smbus.c`, with `utils/smbus.h` containing the API declarations for use by applications.

## 14.2 API Functions

### Data Structures

- [tSMBus](#)
- [tSMBusUDID](#)

### Functions

- void [SMBusARPDIsable](#) (tSMBus \*psSMBus)
- void [SMBusARPEnable](#) (tSMBus \*psSMBus)
- void [SMBusARPUDIDPacketDecode](#) (tSMBusUDID \*pUDID, uint8\_t \*pui8Address, uint8\_t \*pui8Data)
- void [SMBusARPUDIDPacketEncode](#) (tSMBusUDID \*pUDID, uint8\_t ui8Address, uint8\_t \*pui8Data)
- tSMBusStatus [SMBusMasterARPAAssignAddress](#) (tSMBus \*psSMBus, uint8\_t \*pui8Data)
- tSMBusStatus [SMBusMasterARPNotifyMaster](#) (tSMBus \*psSMBus, uint8\_t \*pui8Data)
- tSMBusStatus [SMBusMasterARPPrepareToARP](#) (tSMBus \*psSMBus)
- tSMBusStatus [SMBusMasterBlockProcessCall](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, uint8\_t ui8Command, uint8\_t \*pui8TxData, uint8\_t ui8TxSize, uint8\_t \*pui8RxData)
- tSMBusStatus [SMBusMasterBlockRead](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, uint8\_t ui8Command, uint8\_t \*pui8Data)
- tSMBusStatus [SMBusMasterBlockWrite](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, uint8\_t ui8Command, uint8\_t \*pui8Data, uint8\_t ui8Size)
- tSMBusStatus [SMBusMasterByteReceive](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, uint8\_t \*pui8Data)
- tSMBusStatus [SMBusMasterByteSend](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, uint8\_t ui8Data)
- tSMBusStatus [SMBusMasterByteWordRead](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, uint8\_t ui8Command, uint8\_t \*pui8Data, uint8\_t ui8Size)
- tSMBusStatus [SMBusMasterByteWordWrite](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, uint8\_t ui8Command, uint8\_t \*pui8Data, uint8\_t ui8Size)
- tSMBusStatus [SMBusMasterHostNotify](#) (tSMBus \*psSMBus, uint8\_t ui8OwnSlaveAddress, uint8\_t \*pui8Data)
- tSMBusStatus [SMBusMasterI2CRead](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, uint8\_t \*pui8Data, uint8\_t ui8Size)
- tSMBusStatus [SMBusMasterI2CWrite](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, uint8\_t \*pui8Data, uint8\_t ui8Size)
- tSMBusStatus [SMBusMasterI2CWriteRead](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, uint8\_t \*pui8TxData, uint8\_t ui8TxSize, uint8\_t \*pui8RxData, uint8\_t ui8RxSize)
- void [SMBusMasterInit](#) (tSMBus \*psSMBus, uint32\_t ui32I2CBase, uint32\_t ui32SMBusClock)
- void [SMBusMasterIntEnable](#) (tSMBus \*psSMBus)
- tSMBusStatus [SMBusMasterIntProcess](#) (tSMBus \*psSMBus)
- tSMBusStatus [SMBusMasterProcessCall](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, uint8\_t ui8Command, uint8\_t \*pui8TxData, uint8\_t \*pui8RxData)



- tSMBusStatus [SMBusMasterQuickCommand](#) (tSMBus \*psSMBus, uint8\_t ui8TargetAddress, bool bData)
- void [SMBusPECDisable](#) (tSMBus \*psSMBus)
- void [SMBusPECEnable](#) (tSMBus \*psSMBus)
- uint8\_t [SMBusRxPacketSizeGet](#) (tSMBus \*psSMBus)
- void [SMBusSlaveACKSend](#) (tSMBus \*psSMBus, bool bACK)
- void [SMBusSlaveAddressSet](#) (tSMBus \*psSMBus, uint8\_t ui8AddressNum, uint8\_t ui8SlaveAddress)
- bool [SMBusSlaveARPFlagARGet](#) (tSMBus \*psSMBus)
- void [SMBusSlaveARPFlagARSet](#) (tSMBus \*psSMBus, bool bValue)
- bool [SMBusSlaveARPFlagAVGet](#) (tSMBus \*psSMBus)
- void [SMBusSlaveARPFlagAVSet](#) (tSMBus \*psSMBus, bool bValue)
- void [SMBusSlaveBlockTransferDisable](#) (tSMBus \*psSMBus)
- void [SMBusSlaveBlockTransferEnable](#) (tSMBus \*psSMBus)
- uint8\_t [SMBusSlaveCommandGet](#) (tSMBus \*psSMBus)
- tSMBusStatus [SMBusSlaveDataSend](#) (tSMBus \*psSMBus)
- void [SMBusSlaveI2CDisable](#) (tSMBus \*psSMBus)
- void [SMBusSlaveI2CEnable](#) (tSMBus \*psSMBus)
- void [SMBusSlaveInit](#) (tSMBus \*psSMBus, uint32\_t ui32I2CBase)
- tSMBusStatus [SMBusSlaveIntAddressGet](#) (tSMBus \*psSMBus)
- void [SMBusSlaveIntEnable](#) (tSMBus \*psSMBus)
- tSMBusStatus [SMBusSlaveIntProcess](#) (tSMBus \*psSMBus)
- void [SMBusSlaveManualACKDisable](#) (tSMBus \*psSMBus)
- void [SMBusSlaveManualACKEnable](#) (tSMBus \*psSMBus)
- bool [SMBusSlaveManualACKStatusGet](#) (tSMBus \*psSMBus)
- void [SMBusSlaveProcessCallDisable](#) (tSMBus \*psSMBus)
- void [SMBusSlaveProcessCallEnable](#) (tSMBus \*psSMBus)
- void [SMBusSlaveRxBufferSet](#) (tSMBus \*psSMBus, uint8\_t \*pui8Data, uint8\_t ui8Size)
- void [SMBusSlaveTransferInit](#) (tSMBus \*psSMBus)
- void [SMBusSlaveTxBufferSet](#) (tSMBus \*psSMBus, uint8\_t \*pui8Data, uint8\_t ui8Size)
- void [SMBusSlaveUDIDSet](#) (tSMBus \*psSMBus, tSMBusUDID \*pUDID)
- tSMBusStatus [SMBusStatusGet](#) (tSMBus \*psSMBus)

## 14.2.1 Data Structure Documentation

### 14.2.1.1 tSMBus

**Definition:**

```
typedef struct
{
    tSMBusUDID *pUDID;
    uint32_t ui32I2CBase;
    uint8_t *pui8TxBuffer;
    uint8_t *pui8RxBuffer;
    uint8_t ui8TxSize;
    uint8_t ui8TxIndex;
}
```

```
uint8_t ui8RxSize;  
uint8_t ui8RxIndex;  
uint8_t ui8OwnSlaveAddress;  
uint8_t ui8TargetSlaveAddress;  
uint8_t ui8CurrentCommand;  
uint8_t ui8CalculatedCRC;  
uint8_t ui8ReceivedCRC;  
uint8_t ui8MasterState;  
uint8_t ui8SlaveState;  
uint16_t ui16Flags;  
}  
tSMBus
```

**Members:**

**pUDID** The SMBus Unique Device ID (UDID) for this SMBus instance. If operating as a host, master-only, or on a bus that does not use Address Resolution Protocol (ARP), this is not required. This member can be set via a direct structure access or using the SMBusSlaveInit function. For detailed information about the UDID, refer to the SMBus spec.

**ui32I2CBase** The base address of the I2C master peripheral. This member can be set via a direct structure access or using the SMBusMasterInit or SMBusSlaveInit functions.

**pui8TxBuffer** The address of the data buffer used for transmit operations. For master operations, this member is set by the SMBusMasterxxxx functions that pass a buffer pointer (for example, SMBusMasterBlockWrite). For slave operations, this member can be set via direct structure access or using the SMBusSlaveTxBufferSet function.

**pui8RxBuffer** The address of the data buffer used for receive operations. For master operations, this member is set by the SMBusMasterxxxx functions that pass a buffer pointer (for example, SMBusMasterBlockRead). For slave operations, this member can be set via direct structure access or using the SMBusSlaveRxBufferSet function.

**ui8TxSize** The amount of data to transmit from pui8TxBuffer. For master operations this member is set by the SMBusMasterxxxx functions either via an input argument (example SMBusMasterByteWordWrite) or explicitly (example SMBusMasterSendByte). In master mode, this member should not be accessed or modified by the application. For slave operations, this member can be set via direct structure access or using the SMBusSlaveTxBufferSet function.

**ui8TxIndex** The current index in the transmit buffer. This member should not be accessed or modified by the application.

**ui8RxSize** The amount of data to receive into pui8RxBuffer. For master operations, this member is set by the SMBusMasterxxxx functions either via an input argument (example SMBusMasterByteWordRead), explicitly (example SMBusMasterReceiveByte), or by the slave (example SMBusMasterBlockRead). In master mode, this member should not be accessed or modified by the application. For slave operations, this member can be set via direct structure access or using the SMBusSlaveRxBufferSet function.

**ui8RxIndex** The current index in the receive buffer. This member should not be accessed or modified by the application.

**ui8OwnSlaveAddress** The active slave address of the I2C peripheral on the device. When using dual address in slave mode, the active address is store here. In master mode, this member is not used. This member is updated as requests come in from the master.

**ui8TargetSlaveAddress** The address of the targeted slave device. In master mode, this member is set by the ui8TargetSlaveAddress argument in the SMBusMasterxxxx transfer functions. In slave mode, it is not used. This member should not be modified by the application.

**ui8CurrentCommand** The last used command. In master mode, this member is set by the ui8Command argument in the SMBusMasterxxxx transfer functions. In slave mode, the

first received byte will always be considered the command. This member should not be modified by the application.

**ui8CalculatedCRC** The running CRC calculation used for transfers that require Packet Error Checking (PEC). This member is updated by the SMBus software and should not be modified by the application.

**ui8ReceivedCRC** The received CRC calculation used for transfers that require Packet Error Checking (PEC). This member is updated by the SMBus software and should not be modified by the application.

**ui8MasterState** The current state of the SMBusMasterISRProcess state machine. This member should not be accessed or modified by the application.

**ui8SlaveState** The current state of the SMBusSlaveISRProcess state machine. This member should not be accessed or modified by the application.

**ui16Flags** Flags used for various items in the SMBus state machines for different transaction types and status.

FLAG\_PEC can be modified via the SMBusPECEnable or SMBusPECDisable functions or via direct structure access.

FLAG\_BLOCK\_TRANSFER can be set via the SMBusSlaveBlockTransferEnable function and is cleared automatically by the SMBusSlaveTransferInit function or manually using the SMBusSlaveBlockTransferDisable function.

FLAG\_RAW\_I2C can be modified via the SMBusSlaveI2CEnable or SMBusSlaveI2CDisable functions or via direct structure access.

FLAG\_TRANSFER\_IN\_PROGRESS should not be modified by the application, but can be read via the SMBusStatusGet function.

FLAG\_PROCESS\_CALL can be set via the SMBusSlaveProcessCallEnable function and is cleared automatically by the SMBusSlaveTransferInit function or manually using the SMBusSlaveProcessCallDisable function.

FLAG\_ADDRESS\_RESOLVED is only used by an SMBus Slave that supports ARP. This flag can be modified via the SMBusSlaveARPFlagARSet function and read via SMBusSlaveARPFlagARGet. It can also be modified by direct structure access.

FLAG\_ADDRESS\_VALID is only used by an SMBus Slave that supports ARP. This flag can be modified via the SMBusSlaveARPFlagAVSet function and read via SMBusSlaveARPFlagAVGet. It can also be modified by direct structure access.

FLAG\_ARP is used to indicate that ARP is currently active. This flag is not used by the SMBus stack and can (optionally) be used by the application to keep track of the ARP session.

#### Description:

This structure contains the state of a single instance of an SMBus module. Master and slave instances require unique configuration structures.

### 14.2.1.2 tSMBusUDID

#### Definition:

```
typedef struct
{
    uint8_t ui8DeviceCapabilities;
    uint8_t ui8Version;
    uint16_t ui16VendorID;
    uint16_t ui16DeviceID;
    uint16_t ui16Interface;
    uint16_t ui16SubSystemVendorID;
```

```
uint16_t ui16SubSystemDeviceID;  
uint32_t ui32VendorSpecificID;  
}  
tSMBusUDID
```

**Members:**

**ui8DeviceCapabilities** Device capabilities field. This 8-bit field reports generic SMBus capabilities such as address type for ARP.

**ui8Version** Version Revision field. This 8-bit field reports UDID revision information as well as some vendor-specific things such as silicon revision.

**ui16VendorID** Vendor ID. This 16-bit field contains the manufacturer's ID as assigned by the SBS Implementers' Forum of the PCI SIG.

**ui16DeviceID** Device ID. This 16-bit field contains the device ID assigned by the device manufacturer.

**ui16Interface** Interface. This 16-bit field identifies the protocol layer interfaces supported over the SMBus connection.

**ui16SubSystemVendorID** Subsystem Vendor ID. This 16-bit field holds additional information that may be derived from the vendor ID or other information.

**ui16SubSystemDeviceID** Subsystem Device ID. This 16-bit field holds additional information that may be derived from the device ID or other information.

**ui32VendorSpecificID** Vendor-specific ID. This 32-bit field contains a unique number that can be assigned per device by the manufacturer.

**Description:**

This structure holds the SMBus Unique Device ID (UDID). For detailed information, please refer to the SMBus Specification.

## 14.2.2 Function Documentation

### 14.2.2.1 SMBusARPDisable

Clears the ARP flag in the configuration structure.

**Prototype:**

```
void  
SMBusARPDisable(tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function clears the Address Resolution Protocol (ARP) flag in the configuration structure. This flag can be used to track the state of a device during the ARP process.

**Returns:**

None.

### 14.2.2.2 SMBusARPEnable

Sets the ARP flag in the configuration structure.

**Prototype:**

```
void  
SMBusARPEnable (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function sets the Address Resolution Protocol (ARP) flag in the configuration structure. This flag can be used to track the state of a device during the ARP process.

**Returns:**

None.

### 14.2.2.3 SMBusARPUDIDPacketDecode

Decodes an SMBus packet into a UDID structure and address.

**Prototype:**

```
void  
SMBusARPUDIDPacketDecode (tSMBusUDID *pUDID,  
                           uint8_t *pui8Address,  
                           uint8_t *pui8Data)
```

**Parameters:**

**pUDID** specifies the structure that is updated with new data.

**pui8Address** specifies the location of the variable that holds the the address sent with the UDID (byte 17).

**pui8Data** specifies the location of the source data.

**Description:**

This function takes a data buffer and decodes it into a tSMBusUDID structure and an address variable. It is assumed that there are 17 bytes in the data buffer.

**Returns:**

None.

### 14.2.2.4 SMBusARPUDIDPacketEncode

Encodes a UDID structure and address into SMBus-transferable byte order.

**Prototype:**

```
void  
SMBusARPUDIDPacketEncode (tSMBusUDID *pUDID,  
                           uint8_t ui8Address,  
                           uint8_t *pui8Data)
```

**Parameters:**

**pUDID** specifies the structure to encode.

**ui8Address** specifies the address to send with the UDID (byte 17).

***pui8Data*** specifies the location of the destination data buffer.

**Description:**

This function takes a [tSMBusUDID](#) structure and re-orders the bytes so that it can be transferred on the bus. The destination data buffer must contain at least 17 bytes.

**Returns:**

None.

#### 14.2.2.5 SMBusMasterARPAAssignAddress

Sends an ARP Assign Address packet.

**Prototype:**

```
tSMBusStatus  
SMBusMasterARPAAssignAddress(tSMBus *psSMBus,  
                             uint8_t *pui8Data)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

***pui8Data*** is a pointer to the transmit data buffer. This buffer should be correctly formatted using [SMBusARPUIDPacketEncode\(\)](#) and should contain the UDID data and the address for the slave.

**Description:**

This function sends an Assign Address packet, used during Address Resolution Protocol (ARP). Because SMBus requires data bytes be sent out MSB first, the UDID and target address should be formatted correctly by the application or using [SMBusARPUIDPacketEncode\(\)](#) and placed into a data buffer pointed to by *pui8Data*.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.6 SMBusMasterARPNotifyMaster

Sends a Notify ARP Master packet.

**Prototype:**

```
tSMBusStatus  
SMBusMasterARPNotifyMaster(tSMBus *psSMBus,  
                           uint8_t *pui8Data)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

***pui8Data*** is a pointer to the transmit data buffer. The data payload should be 0x0000 for this packet.

**Description:**

This function sends a Notify ARP Master packet, used during Address Resolution Protocol (ARP). This packet is used by a slave to indicate to the ARP Master that it needs attention.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.7 SMBusMasterARPPrepareToARP

Sends a Prepare to ARP packet.

**Prototype:**

```
tSMBusStatus
SMBusMasterARPPrepareToARP (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function sends a Prepare to ARP packet, used during Address Resolution Protocol (ARP). This packet is used by an ARP Master to alert devices on the bus that ARP is about to begin. All ARP-capable devices must acknowledge all bytes in this packet and clear their Address Resolved (AR) flag.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.8 SMBusMasterBlockProcessCall

Initiates a master Block Process Call transfer to an SMBus slave.

**Prototype:**

```
tSMBusStatus
SMBusMasterBlockProcessCall (tSMBus *psSMBus,
                             uint8_t ui8TargetAddress,
                             uint8_t ui8Command,
                             uint8_t *pui8TxData,
                             uint8_t ui8TxSize,
                             uint8_t *pui8RxData)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8TargetAddress** specifies the slave address of the target device.

**ui8Command** is the command byte sent before the data is requested.

**pui8TxData** is a pointer to the transmit data buffer.

**ui8TxSize** is the number of bytes to send to the slave.

***pui8RxData*** is a pointer to the receive data buffer.

**Description:**

This function supports the Block Write/Block Read Process Call protocol. The amount of data sent to the slave is user defined but limited to 32 data bytes. The amount of data read is defined by the slave device, but should never exceed 32 bytes per the SMBus spec. The receive size is the first data byte returned by the slave, so the actual size is populated in `SMBusMasterISRProcess()`. In the application interrupt handler, [SMBusRxPacketSizeGet\(\)](#) can be used to obtain the amount of data sent by the slave.

This protocol supports the optional PEC byte for error checking. To use PEC, [SMBusPECEnable\(\)](#) must be called before this function.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUS\_BUSY** if the bus is already in use, **SMBUS\_DATA\_SIZE\_ERROR** if `ui8TxSize` is greater than 32, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.9 SMBusMasterBlockRead

Initiates a master Block Read transfer to an SMBus slave.

**Prototype:**

```
tSMBusStatus  
SMBusMasterBlockRead(tSMBus *psSMBus,  
                     uint8_t ui8TargetAddress,  
                     uint8_t ui8Command,  
                     uint8_t *pui8Data)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

***ui8TargetAddress*** specifies the slave address of the target device.

***ui8Command*** is the command byte sent before the data is requested.

***pui8Data*** is a pointer to the receive data buffer.

**Description:**

This function supports the Block Read protocol. The amount of data read is defined by the slave device, but should never exceed 32 bytes per the SMBus spec. The receive size is the first data byte returned by the slave, so this function assumes a size of 3 until the actual number is sent by the slave. In the application interrupt handler, [SMBusRxPacketSizeGet\(\)](#) can be used to obtain the amount of data sent by the slave.

This protocol supports the optional PEC byte for error checking. To use PEC, [SMBusPECEnable\(\)](#) must be called before this function.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.



### 14.2.2.10 SMBusMasterBlockWrite

Initiates a master Block Write transfer to an SMBus slave.

**Prototype:**

```
tSMBusStatus
SMBusMasterBlockWrite(tSMBus *psSMBus,
                      uint8_t ui8TargetAddress,
                      uint8_t ui8Command,
                      uint8_t *pui8Data,
                      uint8_t ui8Size)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8TargetAddress** specifies the slave address of the target device.

**ui8Command** is the command byte sent before the data is requested.

**pui8Data** is a pointer to the transmit data buffer.

**ui8Size** is the number of bytes to send to the slave.

**Description:**

This function supports the Block Write protocol. The amount of data sent to the slave is user defined, but limited to 32 bytes per the SMBus spec.

This protocol supports the optional PEC byte for error checking. To use PEC, [SMBusPECEnable\(\)](#) must be called before this function.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUSY** if the bus is already in use, **SMBUS\_DATA\_SIZE\_ERROR** if ui8Size is greater than 32, or **SMBUS\_OK** if the transfer has successfully been initiated.

### 14.2.2.11 SMBusMasterByteReceive

Initiates a master Receive Byte transfer to an SMBus slave.

**Prototype:**

```
tSMBusStatus
SMBusMasterByteReceive(tSMBus *psSMBus,
                      uint8_t ui8TargetAddress,
                      uint8_t *pui8Data)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8TargetAddress** specifies the slave address of the target device.

**pui8Data** is a pointer to the location to store the received data byte.

**Description:**

The Receive Byte protocol is a basic SMBus protocol that receives a single data byte from the slave. Unlike most of the other SMBus protocols, Receive Byte does not send a “command” byte before the data payload and is intended for basic communication.

This protocol supports the optional PEC byte for error checking. To use PEC, [SMBusPECEnable\(\)](#) must be called before this function.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.12 SMBusMasterByteSend

Initiates a master Send Byte transfer to an SMBus slave.

**Prototype:**

```
tSMBusStatus  
SMBusMasterByteSend(tSMBus *psSMBus,  
                    uint8_t ui8TargetAddress,  
                    uint8_t ui8Data)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8TargetAddress** specifies the slave address of the target device.

**ui8Data** is the data byte to send to the slave.

**Description:**

The Send Byte protocol is a basic SMBus protocol that sends a single data byte to the slave. Unlike most of the other SMBus protocols, Send Byte does not send a “command” byte before the data payload and is intended for basic communication.

This protocol supports the optional PEC byte for error checking. To use PEC, [SMBusPECEnable\(\)](#) must be called before this function.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.13 SMBusMasterByteWordRead

Initiates a master Read Byte or Read Word transfer to an SMBus slave.

**Prototype:**

```
tSMBusStatus  
SMBusMasterByteWordRead(tSMBus *psSMBus,  
                        uint8_t ui8TargetAddress,  
                        uint8_t ui8Command,  
                        uint8_t *pui8Data,  
                        uint8_t ui8Size)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8TargetAddress** specifies the slave address of the target device.

**ui8Command** is the command byte sent before the data is requested.

**pui8Data** is a pointer to the receive data buffer.

**ui8Size** is the number of bytes to receive from the slave.

**Description:**

This function supports both the Read Byte and Read Word protocols. The amount of data to receive is user defined, but limited to 1 or 2 bytes.

This protocol supports the optional PEC byte for error checking. To use PEC, [SMBusPECEnable\(\)](#) must be called before this function.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUS\_BUSY** if the bus is already in use, **SMBUS\_DATA\_SIZE\_ERROR** if ui8Size is greater than 2, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.14 SMBusMasterByteWordWrite

Initiates a master Write Byte or Write Word transfer to an SMBus slave.

**Prototype:**

```
tSMBusStatus  
SMBusMasterByteWordWrite(tSMBus *psSMBus,  
                          uint8_t ui8TargetAddress,  
                          uint8_t ui8Command,  
                          uint8_t *pui8Data,  
                          uint8_t ui8Size)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8TargetAddress** specifies the slave address of the target device.

**ui8Command** is the command byte sent before the data payload.

**pui8Data** is a pointer to the transmit data buffer.

**ui8Size** is the number of bytes to send to the slave.

**Description:**

This function supports both the Write Byte and Write Word protocols. The amount of data to send is user defined, but limited to 1 or 2 bytes.

This protocol supports the optional PEC byte for error checking. To use PEC, [SMBusPECEnable\(\)](#) must be called before this function.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUS\_BUSY** if the bus is already in use, **SMBUS\_DATA\_SIZE\_ERROR** if ui8Size is greater than 2, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.15 SMBusMasterHostNotify

Initiates a master Host Notify transfer to the SMBus Host.

**Prototype:**

```
tSMBusStatus  
SMBusMasterHostNotify(tSMBus *psSMBus,  
                      uint8_t ui8OwnSlaveAddress,  
                      uint8_t *pui8Data)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8OwnSlaveAddress** specifies the peripheral's own slave address.

**pui8Data** is a pointer to the two byte data payload.

**Description:**

The Host Notify protocol is used by SMBus slaves to alert the bus Host about an event. Most slave devices that operate in this environment only become a bus master when this packet type is used. Host Notify always sends two data bytes to the host along with the peripheral's own slave address so that the Host knows which peripheral requested the Host's attention.

This protocol does not support PEC. The PEC flag is explicitly cleared within this function, so if PEC is enabled prior to calling it, it must be re-enabled afterwards.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.16 SMBusMasterI2CRead

Initiates a "raw" I2C read transfer to a slave device.

**Prototype:**

```
tSMBusStatus  
SMBusMasterI2CRead(tSMBus *psSMBus,  
                  uint8_t ui8TargetAddress,  
                  uint8_t *pui8Data,  
                  uint8_t ui8Size)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8TargetAddress** specifies the slave address of the target device.

**pui8Data** is a pointer to the receive data buffer.

**ui8Size** is the number of bytes to send to the slave.

**Description:**

This function receives a user-defined number of bytes from an I2C slave without using an SMBus protocol. The data size is only limited to the size of the ui8Size variable, which is an unsigned character (8 bits, value of 255).

Because this function uses "raw" I2C, PEC is not supported.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.

### 14.2.2.17 SMBusMasterI2CWrite

Initiates a “raw” I2C write transfer to a slave device.

**Prototype:**

```
tSMBusStatus  
SMBusMasterI2CWrite(tSMBus *psSMBus,  
                    uint8_t ui8TargetAddress,  
                    uint8_t *pui8Data,  
                    uint8_t ui8Size)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8TargetAddress** specifies the slave address of the target device.

**pui8Data** is a pointer to the transmit data buffer.

**ui8Size** is the number of bytes to send to the slave.

**Description:**

This function sends a user-defined number of bytes to an I2C slave without using an SMBus protocol. The data size is only limited to the size of the ui8Size variable, which is an unsigned character (8 bits, value of 255).

Because this function uses “raw” I2C, PEC is not supported.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.

### 14.2.2.18 SMBusMasterI2CWriteRead

Initiates a “raw” I2C write-read transfer to a slave device.

**Prototype:**

```
tSMBusStatus  
SMBusMasterI2CWriteRead(tSMBus *psSMBus,  
                        uint8_t ui8TargetAddress,  
                        uint8_t *pui8TxData,  
                        uint8_t ui8TxSize,  
                        uint8_t *pui8RxData,  
                        uint8_t ui8RxSize)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8TargetAddress** specifies the slave address of the target device.

**pui8TxData** is a pointer to the transmit data buffer.

**ui8TxSize** is the number of bytes to send to the slave.

**pui8RxData** is a pointer to the receive data buffer.

**ui8RxSize** is the number of bytes to receive from the slave.

**Description:**

This function initiates a write-read transfer to an I2C slave without using an SMBus protocol. The user-defined number of bytes is written to the slave first, followed by the reception of the user-defined number of bytes. The transmit and receive data sizes are only limited to the size of the `ui8TxSize` and `ui8RxSize` variables, which are unsigned characters (8 bits, value of 255).

Because this function uses “raw” I2C, PEC is not supported.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.19 SMBusMasterInit

Initializes an I2C master peripheral for SMBus functionality.

**Prototype:**

```
void
SMBusMasterInit(tSMBus *psSMBus,
                uint32_t ui32I2CBase,
                uint32_t ui32SMBusClock)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui32I2CBase** specifies the base address of the I2C master peripheral.

**ui32SMBusClock** specifies the system clock speed of the MCU.

**Description:**

This function initializes an I2C peripheral for SMBus master use. The instance-specific configuration structure is initialized to a set of known values and the I2C peripheral is configured for 100kHz use, which is required by the SMBus specification.

**Returns:**

None.

#### 14.2.2.20 SMBusMasterIntEnable

Enables the appropriate master interrupts for stack processing.

**Prototype:**

```
void
SMBusMasterIntEnable(tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function enables the I2C interrupts used by the SMBus master. Both the peripheral-level and NVIC-level interrupts are enabled. [SMBusMasterInit\(\)](#) must be called before this function because this function relies on the I2C base address being defined.

**Returns:**

None.

### 14.2.2.21 SMBusMasterIntProcess

Master ISR processing function for the SMBus application.

**Prototype:**

```
tSMBusStatus
SMBusMasterIntProcess (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function must be called in the application interrupt service routine (ISR) to process SMBus master interrupts.

**Returns:**

Returns **SMBUS\_TIMEOUT** if a bus timeout is detected, **SMBUS\_ARB\_LOST** if I2C bus arbitration lost is detected, **SMBUS\_ADDR\_ACK\_ERROR** if the address phase of a transfer results in a NACK, **SMBUS\_DATA\_ACK\_ERROR** if the data phase of a transfer results in a NACK, **SMBUS\_DATA\_SIZE\_ERROR** if a receive buffer overrun is detected or if a transmit operation tries to write more data than is allowed, **SMBUS\_MASTER\_ERROR** if an unknown error occurs, **SMBUS\_PEC\_ERROR** if the received PEC byte does not match the locally calculated value, or **SMBUS\_OK** if processing finished successfully.

### 14.2.2.22 SMBusMasterProcessCall

Initiates a master Process Call transfer to an SMBus slave.

**Prototype:**

```
tSMBusStatus
SMBusMasterProcessCall (tSMBus *psSMBus,
                        uint8_t ui8TargetAddress,
                        uint8_t ui8Command,
                        uint8_t *pui8TxData,
                        uint8_t *pui8RxData)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8TargetAddress** specifies the slave address of the target device.

**ui8Command** is the command byte sent before the data is requested.

**pui8TxData** is a pointer to the transmit data buffer.

**pui8RxData** is a pointer to the receive data buffer.

**Description:**

This function supports the Process Call protocol. The amount of data sent to and received from the slave is fixed to 2 bytes per direction (2 sent, 2 received).

This protocol supports the optional PEC byte for error checking. To use PEC, [SMBusPECEnable\(\)](#) must be called before this function.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.23 SMBusMasterQuickCommand

Initiates a master Quick Command transfer to an SMBus slave.

**Prototype:**

```
tSMBusStatus  
SMBusMasterQuickCommand(tSMBus *psSMBus,  
                        uint8_t ui8TargetAddress,  
                        bool bData)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8TargetAddress** specifies the slave address of the target device.

**bData** is the value of the single data bit sent to the slave.

**Description:**

Quick Command is an SMBus protocol that sends a single data bit using the I2C R/S bit. This function issues a single I2C transfer with the slave address and data bit.

This protocol does not support PEC. The PEC flag is explicitly cleared within this function, so if PEC is enabled prior to calling it, it must be re-enabled afterwards.

**Returns:**

Returns **SMBUS\_PERIPHERAL\_BUSY** if the I2C peripheral is currently active, **SMBUS\_BUS\_BUSY** if the bus is already in use, or **SMBUS\_OK** if the transfer has successfully been initiated.

#### 14.2.2.24 SMBusPECDisable

Disables Packet Error Checking (PEC).

**Prototype:**

```
void  
SMBusPECDisable(tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function disables the transmission and checking of a PEC byte in SMBus transactions.

**Returns:**

None.



#### 14.2.2.25 SMBusPECEnable

Enables Packet Error Checking (PEC).

**Prototype:**

```
void  
SMBusPECEnable (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function enables the transmission and checking of a PEC byte in SMBus transactions.

**Returns:**

None.

#### 14.2.2.26 SMBusRxPacketSizeGet

Returns the number of bytes in the receive buffer.

**Prototype:**

```
uint8_t  
SMBusRxPacketSizeGet (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function returns the number of bytes in the active receive buffer. It can be used to determine how many bytes have been received in the slave receive or master block read configurations.

**Returns:**

Number of bytes in the buffer.

#### 14.2.2.27 SMBusSlaveACKSend

Sets the value of the ACK bit when using manual acknowledgement.

**Prototype:**

```
void  
SMBusSlaveACKSend (tSMBus *psSMBus,  
                   bool bACK)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**bACK** specifies whether to ACK (**true**) or NACK (**false**).

**Description:**

This function sets the value of the ACK bit. In order for the ACK bit to take effect, manual acknowledgement must be enabled on the slave using [SMBusSlaveManualACKEnable\(\)](#).

**Returns:**

None.

#### 14.2.2.28 SMBusSlaveAddressSet

Sets the slave address for an SMBus slave peripheral.

**Prototype:**

```
void  
SMBusSlaveAddressSet (tSMBus *psSMBus,  
                      uint8_t ui8AddressNum,  
                      uint8_t ui8SlaveAddress)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**ui8AddressNum** specifies which address (primary or secondary)

**ui8SlaveAddress** is the address of the slave.

**Description:**

This function sets the slave address. Both the primary and secondary addresses can be set using this function. To set the primary address (stored in I2CSOAR), ui8AddressNum should be '0'. To set the secondary address (stored in I2CSOAR2), ui8AddressNum should be '1'.

**Returns:**

None.

#### 14.2.2.29 SMBusSlaveARPFflagARGet

Returns the current value of the AR (Address Resolved) flag.

**Prototype:**

```
bool  
SMBusSlaveARPFflagARGet (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This returns the value of the AR (Address Resolved) flag.

**Returns:**

Returns **true** if set, **false** if cleared.

#### 14.2.2.30 SMBusSlaveARPFflagARSet

Sets the value of the AR (Address Resolved) flag.

**Prototype:**

```
void  
SMBusSlaveARPFlagARSet (tSMBus *psSMBus,  
                        bool bValue)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.  
**bValue** is the value to set the flag.

**Description:**

This function allows the application to set the value of the AR flag. All SMBus slaves must support the AR and AV flags. On POR, the AR flag is cleared. It is also cleared when a slave receives the ARP Reset Device command.

**Returns:**

None.

#### 14.2.2.31 SMBusSlaveARPFlagAVGet

Returns the current value of the AV (Address Valid) flag.

**Prototype:**

```
bool  
SMBusSlaveARPFlagAVGet (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This returns the value of the AV (Address Valid) flag.

**Returns:**

Returns **true** if set, or **false** if cleared.

#### 14.2.2.32 SMBusSlaveARPFlagAVSet

Sets the value of the AV (Address Valid) flag.

**Prototype:**

```
void  
SMBusSlaveARPFlagAVSet (tSMBus *psSMBus,  
                        bool bValue)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.  
**bValue** is the value to set the flag.

**Description:**

This function allows the application to set the value of the AV flag. All SMBus slaves must support the AR and AV flags. On POR, the AV flag is cleared. It is also cleared when a slave receives the ARP Reset Device command.

**Returns:**

None.

#### 14.2.2.33 SMBusSlaveBlockTransferDisable

Clears the block transfer flag for an SMBus slave transfer.

**Prototype:**

```
void  
SMBusSlaveBlockTransferDisable(tSMBus *psSMBus)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

**Description:**

Clears the block transfer flag in the configuration structure. The user application can either call this function to clear the flag, or use [SMBusSlaveTransferInit\(\)](#) to clear out all transfer-specific flags.

**Returns:**

None.

#### 14.2.2.34 SMBusSlaveBlockTransferEnable

Sets the block transfer flag for an SMBus slave transfer.

**Prototype:**

```
void  
SMBusSlaveBlockTransferEnable(tSMBus *psSMBus)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

**Description:**

Sets the block transfer flag in the configuration structure so that the SMBus slave can respond correctly to a Block Write or Block Read request. This flag must be set prior to the data portion of the packet.

**Returns:**

None.

#### 14.2.2.35 SMBusSlaveCommandGet

Get the current command byte.

**Prototype:**

```
uint8_t  
SMBusSlaveCommandGet(tSMBus *psSMBus)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

**Description:**

Returns the current value of the `ui8CurrentCommand` variable in the SMBus configuration structure. This can be used to help the user application set up the SMBus slave transmit and receive buffers.

**Returns:**

None.

#### 14.2.2.36 SMBusSlaveDataSend

Sends data outside of the interrupt processing function.

**Prototype:**

```
tSMBusStatus  
SMBusSlaveDataSend(tSMBus *psSMBus)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

**Description:**

This function sends data outside the interrupt processing function, and should only be used when [SMBusSlaveIntProcess\(\)](#) returns **SMBUS\_SLAVE\_NOT\_READY**. At this point, the application should set up the transfer and call this function (it assumes that the transmit buffer has already been populated when called). When called, this function updates the slave state machine as if [SMBusSlaveIntProcess\(\)](#) were called.

**Returns:**

Returns **SMBUS\_SLAVE\_NOT\_READY** if the slave's transmit buffer is not yet initialized (`ui8TxSize` is 0), or **SMBUS\_OK** if processing finished successfully.

#### 14.2.2.37 SMBusSlaveI2CDisable

Clears the "raw" I2C flag for an SMBus slave transfer.

**Prototype:**

```
void  
SMBusSlaveI2CDisable(tSMBus *psSMBus)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

**Description:**

Clears the raw I2C flag in the configuration structure. This flag is a global setting similar to the PEC flag and cannot be cleared using [SMBusSlaveTransferInit\(\)](#).

**Returns:**

None.

#### 14.2.2.38 SMBusSlaveI2CEnable

Sets the “raw” I2C flag for an SMBus slave transfer.

**Prototype:**

```
void  
SMBusSlaveI2CEnable(tSMBus *psSMBus)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

**Description:**

Sets the raw I2C flag in the configuration structure so that the SMBus slave can respond correctly to raw I2C (non-SMBus protocol) requests. This flag must be set prior to the transfer, and is a global setting.

**Returns:**

None.

#### 14.2.2.39 SMBusSlaveInit

Initializes an I2C slave peripheral for SMBus functionality.

**Prototype:**

```
void  
SMBusSlaveInit(tSMBus *psSMBus,  
               uint32_t ui32I2CBase)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

***ui32I2CBase*** specifies the base address of the I2C slave peripheral.

**Description:**

This function initializes an I2C peripheral for SMBus slave use. The instance-specific configuration structure is initialized to a set of known values and the I2C peripheral is configured based on the input arguments.

The default configuration of the SMBus slave uses automatic acknowledgement. If manual acknowledgement is required, call [SMBusSlaveManualACKEnable\(\)](#).

**Returns:**

None.

#### 14.2.2.40 SMBusSlaveIntAddressGet

Determine whether primary or secondary slave address has been requested by the master.

**Prototype:**

```
tSMBusStatus  
SMBusSlaveIntAddressGet(tSMBus *psSMBus)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

**Description:**

Tells the caller whether the I2C slave address requested by the master or SMBus Host is the primary or secondary I2C slave address of the peripheral. The primary is defined as the address programmed into I2CSOAR, and the secondary as the address programmed into I2CSOAR2.

**Returns:**

Returns **SMBUS\_SLAVE\_ADDR\_PRIMARY** if the primary address is called out or **SMBUS\_SLAVE\_ADDR\_SECONDARY** if the secondary address is called out.

#### 14.2.2.41 SMBusSlaveIntEnable

Enables the appropriate slave interrupts for stack processing.

**Prototype:**

```
void  
SMBusSlaveIntEnable(tSMBus *psSMBus)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

**Description:**

This function enables the I2C interrupts used by the SMBus slave. Both the peripheral-level and NVIC-level interrupts are enabled. [SMBusSlaveInit\(\)](#) must be called before this function because this function relies on the I2C base address being defined.

**Returns:**

None.

#### 14.2.2.42 SMBusSlaveIntProcess

Slave ISR processing function for the SMBus application.

**Prototype:**

```
tSMBusStatus  
SMBusSlaveIntProcess(tSMBus *psSMBus)
```

**Parameters:**

***psSMBus*** specifies the SMBus configuration structure.

**Description:**

This function must be called in the application interrupt service routine (ISR) to process SMBus slave interrupts.

If manual acknowledge is enabled using [SMBusSlaveManualACKEnable\(\)](#), this function processes the data byte, but does not send the ACK/NACK value. In this case, the user application is responsible for sending the acknowledge bit based on the return code of this function.

When receiving a Quick Command from the master, the slave has some set-up requirements. When the master sends the R/S (data) bit as '0', nothing additional needs to be done in the

slave and `SMBusSlaveIntProcess()` returns **SMBUS\_SLAVE\_QCMD\_0**. However, when the master sends the R/S (data) bit as '1', the slave must write the data register with data containing a '1' in bit 7. This means that when receiving a Quick Command, the slave must set up the TX buffer to either have 1 data byte with bit 7 set to '1' or set up the TX buffer to be zero length. In the case where 1 data byte is put in the TX buffer, `SMBusSlaveIntProcess()` returns **SMBUS\_OK** the first time its called and **SMBUS\_SLAVE\_QCMD\_0** the second. In the case where the TX buffer has no data, `SMBusSlaveIntProcess()` will return **SMBUS\_SLAVE\_ERROR** the first time its called, and **SMBUS\_SLAVE\_QCMD\_1** the second time.

**Returns:**

Returns **SMBUS\_SLAVE\_FIRST\_BYTE** if the first byte (typically the SMBus command) has been received; **SMBUS\_SLAVE\_NOT\_READY** if the slave's transmit buffer is not yet initialized when the master requests data from the slave; **SMBUS\_DATA\_SIZE\_ERROR** if during a master block write, the size sent by the master is greater than the amount of available space in the receive buffer; **SMBUS\_SLAVE\_ERROR** if a buffer overrun is detected during a slave receive operation or if data is sent and was not expected; **SMBUS\_SLAVE\_QCMD\_0** if a Quick Command was received with data '0'; **SMBUS\_SLAVE\_QCMD\_1** if a Quick Command was received with data '1'; **SMBUS\_TRANSFER\_COMPLETE** if a STOP is detected on the bus, marking the end of a transfer; **SMBUS\_PEC\_ERROR** if the received PEC byte does not match the locally calculated value; or **SMBUS\_OK** if processing finished successfully.

#### 14.2.2.43 SMBusSlaveManualACKDisable

Disables manual acknowledgement for the SMBus slave.

**Prototype:**

```
void  
SMBusSlaveManualACKDisable (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function disables manual acknowledge capability in the slave. When manual acknowledgement is disabled, the slave automatically ACKs every byte sent by the master.

**Returns:**

None.

#### 14.2.2.44 SMBusSlaveManualACKEnable

Enables manual acknowledgement for the SMBus slave.

**Prototype:**

```
void  
SMBusSlaveManualACKEnable (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.



**Description:**

This function enables manual acknowledge capability in the slave. If the application requires that the slave NACK on a bad command or a bad PEC calculation, manual acknowledgement allows this to happen.

In the case of responding to a bad command with a NACK, the application should use [SMBusSlaveACKSend\(\)](#) to ACK/NACK the command. The slave ISR should check for the SMBUS\_SLAVE\_FIRST\_BYTE return code from `SMBusSlaveISRProcess()` and ACK/NACK accordingly. All other cases should be handled in the application based on the return code of `SMBusSlaveISRProcess()`.

**Returns:**

None.

#### 14.2.2.45 SMBusSlaveManualACKStatusGet

Returns the manual acknowledgement status of the SMBus slave.

**Prototype:**

```
bool  
SMBusSlaveManualACKStatusGet (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function returns the state of the I2C ACKOEN bit in the I2CSACKCTL register. This feature is disabled out of reset and must be enabled using [SMBusSlaveManualACKEnable\(\)](#).

**Returns:**

Returns **true** if manual acknowledge is enabled, or **false** if manual acknowledge is disabled.

#### 14.2.2.46 SMBusSlaveProcessCallDisable

Clears the process call flag for an SMBus slave transfer.

**Prototype:**

```
void  
SMBusSlaveProcessCallDisable (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

Clears the process call flag in the configuration structure. The user application can either call this function to clear the flag, or use [SMBusSlaveTransferInit\(\)](#) to clear out all transfer-specific flags.

**Returns:**

None.

#### 14.2.2.47 SMBusSlaveProcessCallEnable

Sets the process call flag for an SMBus slave transfer.

**Prototype:**

```
void  
SMBusSlaveProcessCallEnable (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

Sets the process call flag in the configuration structure so that the SMBus slave can respond correctly to a Process Call request. This flag must be set prior to the data portion of the packet.

**Returns:**

None.

#### 14.2.2.48 SMBusSlaveRxBufferSet

Set the address and size of the slave receive buffer.

**Prototype:**

```
void  
SMBusSlaveRxBufferSet (tSMBus *psSMBus,  
                        uint8_t *pui8Data,  
                        uint8_t ui8Size)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**pui8Data** is a pointer to the receive data buffer.

**ui8Size** is the number of bytes in the buffer.

**Description:**

This function sets the address and size of the slave receive buffer.

**Returns:**

None.

#### 14.2.2.49 SMBusSlaveTransferInit

Sets up the SMBus slave for a new transfer.

**Prototype:**

```
void  
SMBusSlaveTransferInit (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function is used to re-initialize the configuration structure for a new transfer. Once a transfer is complete and the data has been processed, unused flags, states, the data buffers and buffer indexes should be reset to a known state before a new transfer.

**Returns:**

None.

#### 14.2.2.50 SMBusSlaveTxBufferSet

Set the address and size of the slave transmit buffer.

**Prototype:**

```
void  
SMBusSlaveTxBufferSet (tSMBus *psSMBus,  
                        uint8_t *pui8Data,  
                        uint8_t ui8Size)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**pui8Data** is a pointer to the transmit data buffer.

**ui8Size** is the number of bytes in the buffer.

**Description:**

This function sets the address and size of the slave transmit buffer.

**Returns:**

None.

#### 14.2.2.51 SMBusSlaveUDIDSet

Sets a slave's UDID structure.

**Prototype:**

```
void  
SMBusSlaveUDIDSet (tSMBus *psSMBus,  
                   tSMBusUDID *pUDID)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**pUDID** is a pointer to the UDID configuration for the slave. This is only needed if the slave is on a bus that uses ARP.

**Description:**

This function sets the UDID for a slave instance.

**Returns:**

None.

### 14.2.2.52 SMBusStatusGet

Returns the state of an SMBus transfer.

**Prototype:**

```
tSMBusStatus  
SMBusStatusGet (tSMBus *psSMBus)
```

**Parameters:**

**psSMBus** specifies the SMBus configuration structure.

**Description:**

This function returns the status of an SMBus transaction. It can be used to determine whether a transfer is ongoing or complete.

**Returns:**

Returns **SMBUS\_TRANSFER\_IN\_PROGRESS** if transfer is ongoing, or **SMBUS\_TRANSFER\_COMPLETE** if transfer has completed.

## 14.3 Programming Example

The following example shows how to initialize both a master and slave SMBus instance. In this example, it is assumed the I2C0 is the master and resides on pin PB2 and PB3. I2C1 is the slave and resides on pins PA6 and PA7.

```
tSMBus g_sMaster;  
tSMBus g_sSlave;  
unsigned char g_pucSlaveTxBuffer[MAX_SMB_BLOCK_SIZE] = {0};  
unsigned char g_pucSlaveRxBuffer[MAX_SMB_BLOCK_SIZE] = {0};  
  
void  
MyInit(void)  
{  
    //  
    // Enable the peripherals for the SMBus master.  
    //  
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);  
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);  
  
    //  
    // Configure the required pins for I2C0.  
    //  
    GPIOPinTypeI2CSCL(GPIO_PORTB_BASE, GPIO_PIN_2);  
    GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_3);  
  
    //  
    // Configure the IO mux so that the I2C pins for I2C0 are on PB2/3.  
    //  
    GPIOPinConfigure(GPIO_PB2_I2C0SCL);  
    GPIOPinConfigure(GPIO_PB3_I2C0SDA);  
  
    //  
    // Initialize the master SMBus port.  
    //  
    SMBusMasterInit(&g_sMaster, I2C0_BASE, SysCtlClockGet());  
    //  
}
```

```

// Enable master interrupts.
//
SMBusMasterIntEnable(&g_sMaster);

//
// Enable the peripherals for the SMBus slave.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C1);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

//
// Configure the required pins for I2C1.
//
GPIOPinTypeI2CSCL(GPIO_PORTA_BASE, GPIO_PIN_6);
GPIOPinTypeI2C(GPIO_PORTA_BASE, GPIO_PIN_7);

//
// Configure the IO mux so that the I2C pins for I2C1 are on PA6/7.
//
GPIOPinConfigure(GPIO_PA6_I2C1SCL);
GPIOPinConfigure(GPIO_PA7_I2C1SDA);

//
// Initialize the Slave SMBus port.
//
SMBusSlaveInit(&g_sSlave, I2C1_BASE);

//
// Enable slave interrupts.
//
SMBusSlaveIntEnable(&g_sSlave);

//
// Set the slave addresses.
//
SMBusSlaveAddressSet(&g_sSlave, 0, SMBUS_ADR_SMART_BATTERY);

//
// Set up the slave idle transfer. It is good practice to populate the
// slave buffers before attaching to the bus so that they're in a known
// state.
//
SMBusSlaveTxBufferSet(&g_sSlave, g_pucSlaveTxBuffer, MAX_SMB_BLOCK_SIZE);
SMBusSlaveRxBufferSet(&g_sSlave, g_pucSlaveRxBuffer, MAX_SMB_BLOCK_SIZE);
}

```

The following example shows how an application might implement a SMBus master ISR.

```

void
SMBusMasterIntHandler(void)
{
    tSMBusStatus eStatus;

    //
    // Process the interrupt.
    //
    eStatus = SMBusMasterIntProcess(&g_sMaster);

    //
    // Check for errors.
    //
    switch(eStatus)
    {
        case SMBUS_PEC_ERROR:
        {

```

```
        //
        // Handle error.
        //

        break;
    }

    case SMBUS_TIMEOUT:
    {
        //
        // Handle error.
        //

        break;
    }

    case SMBUS_ADDR_ACK_ERROR:
    case SMBUS_DATA_ACK_ERROR:
    {
        //
        // Handle error.
        //

        break;
    }
}
}
```

The following example shows how an application might implement a SMBus slave ISR. Recall that the slave requires much more user application code than the master case. This example shows how to handle various protocols using arbitrary command bytes. Software can either check for the **SMBUS\_TRANSFER\_COMPLETE** return code or use [SMBusStatusGet\(\)](#) to determine when a transfer is complete.

```
void
SMBusSlaveIntProcess(void)
{
    tSMBusStatus eStatus;

    //
    // Process the interrupt.
    //
    eStatus = SMBusSlaveIntProcess(&g_sSlave);

    //
    // See if the first byte/command was received.
    //
    if(eStatus == SMBUS_SLAVE_FIRST_BYTE)
    {
        //
        // Figure out which command was sent and set up the transfer
        // accordingly.
        //
        switch(SMBusSlaveCommandGet(&g_sSlave))
        {
            //
            // Write byte protocol.
            //
            case 0xf1:
            {
                SMBusSlaveRxBufferSet(&g_sSlave, g_pucSlaveRxBuffer, 1);
                break;
            }
        }
    }
}
```

```
//
// Write word protocol.
//
case 0xf2:
{
    SMBusSlaveRxBufferSet(&g_sSlave, g_pucSlaveRxBuffer, 2);
    break;
}

//
// Read byte protocol. TX buffer can be populated in ISR context
// or in another function.
//
case 0xf3:
{
    SMBusSlaveTxBufferSet(&g_sSlave, g_pucSlaveTxBuffer, 1);
    break;
}

//
// Read word protocol. TX buffer can be populated in ISR context
// or in another function.
//
case 0xf4:
{
    SMBusSlaveTxBufferSet(&g_sSlave, g_pucSlaveTxBuffer, 2);
    break;
}

//
// Process call protocol. TX buffer can be populated in ISR
// context or in another function.
//
case 0xf5:
{
    SMBusSlaveProcessCallEnable(&g_sSlave);
    SMBusSlaveTxBufferSet(&g_sSlave, g_pucSlaveTxBuffer, 2);
    SMBusSlaveRxBufferSet(&g_sSlave, g_pucSlaveRxBuffer, 2);
    break;
}

//
// Block write protocol. Set the size to the max to accomodate up
// to the maximum transfer size. When the bus sees the STOP signal
// it marks the end of transfer and the actual size of the RX
// operation is obtained from the configuration structure by using
// SMBusRxPacketSizeGet().
//
case 0xf6:
{
    SMBusSlaveBlockTransferEnable(&g_sSlave);
    SMBusSlaveRxBufferSet(&g_sSlave, g_pucSlaveRxBuffer,
                          MAX_SMB_BLOCK_SIZE);
    break;
}

//
// Block read protocol. The size for the slave TX operation is
// application-specific, so a global variable is used for example
// purpose. TX buffer can be populated in ISR context or in
// another function.
//
case 0xf7:
{
    SMBusSlaveBlockTransferEnable(&g_sSlave);
    SMBusSlaveTxBufferSet(&g_sSlave, g_pucSlaveTxBuffer, g_ucSize);
}
```

```
        break;  
    }  
}  
}
```



# 15 Software I2C Module

Introduction .....	97
API Functions .....	98
Programming Example .....	105

## 15.1 Introduction

The software I2C module uses a timer and two GPIO pins to create a slow-speed software I2C peripheral. Multiple software I2C modules can be in use simultaneously, limited only by the availability of resources (RAM for the per-instance data structure, pins for the interface, timers if unique clock rates are required, and processor cycles to execute the code). The software I2C module supports master mode only; multi-master support is not provided. A callback mechanism is used to simulate the interrupts that would be provided by a hardware I2C module.

The API for the software I2C module has been constructed to be as close as possible to the API provided in the Tiva Peripheral Driver Library for the hardware I2C module. The two notable differences are the function prefix being “SoftI2C” instead of “I2CMaster”, and the first argument of each API is a pointer to the [tSoftI2C](#) data structure instead of the base address of the hardware module.

Timing for the software I2C module is provided by the application. The [SoftI2CTimerTick\(\)](#) function must be called on a periodic basis to provide the timing for the software I2C module. The timer tick function must be called at four times the desired I2C clock rate; for example, to operate the software I2C interface at 10 KHz, the tick function must be called at a 40 KHz rate. By having the application providing the timing, the timer resource can be flexible and multiple software I2C modules can be driven from a single timer resource. Alternatively, if the software I2C module is only needed for brief periods of time and processor usage is not a concern, the timer tick function can simply be called in a loop until the entire I2C transaction has completed (maximizing both I2C clock speed and processor usage, but not requiring a timer).

The software I2C module requires two GPIO pins; one for SCL and one for SDA. The per-instance data structure is approximately 20 bytes in length (the actual length depends on how the structure is packed by the compiler).

As a point of reference, the following are some rough measurements of the processor usage of the software I2C module at various I2C clock speeds with the processor running at 50 MHz. Actual processor usage may vary, depending on how the application uses the software I2C module, processor clock speed, interrupt priority, and compiler.

I2C Clock	% Of Processor	Million Cycles Per Second
5 KHz	4.53	2.26
10 KHz	9.05	4.52
15 KHz	13.53	6.76
20 KHz	18.03	9.01
25 KHz	22.51	11.25
30 KHz	27.05	13.52
35 KHz	31.52	15.76
40 KHz	36.06	18.03
45 KHz	40.54	20.27
50 KHz	44.96	22.48

This module is contained in `utils/softi2c.c`, with `utils/softi2c.h` containing the API declarations for use by applications.

## 15.2 API Functions

### Data Structures

- [tSoftI2C](#)

### Functions

- `bool SoftI2CBusy (tSoftI2C *psl2C)`
- `void SoftI2CCallbackSet (tSoftI2C *psl2C, void (*pfnCallback)(void))`
- `void SoftI2CControl (tSoftI2C *psl2C, uint32_t ui32Cmd)`
- `uint32_t SoftI2CDataGet (tSoftI2C *psl2C)`
- `void SoftI2CDataPut (tSoftI2C *psl2C, uint8_t ui8Data)`
- `uint32_t SoftI2CErr (tSoftI2C *psl2C)`
- `void SoftI2CInit (tSoftI2C *psl2C)`
- `void SoftI2CIntClear (tSoftI2C *psl2C)`
- `void SoftI2CIntDisable (tSoftI2C *psl2C)`
- `void SoftI2CIntEnable (tSoftI2C *psl2C)`
- `bool SoftI2CIntStatus (tSoftI2C *psl2C, bool bMasked)`
- `void SoftI2CSCLGPIOSet (tSoftI2C *psl2C, uint32_t ui32Base, uint8_t ui8Pin)`
- `void SoftI2CSDAGPIOSet (tSoftI2C *psl2C, uint32_t ui32Base, uint8_t ui8Pin)`
- `void SoftI2CSlaveAddrSet (tSoftI2C *psl2C, uint8_t ui8SlaveAddr, bool bReceive)`
- `void SoftI2CTimerTick (tSoftI2C *psl2C)`

### 15.2.1 Data Structure Documentation

#### 15.2.1.1 tSoftI2C

**Definition:**

```
typedef struct
{
    void (*pfnIntCallback) (void);
    uint32_t ui32SCLGPIO;
    uint32_t ui32SDAGPIO;
    uint8_t ui8Flags;
    uint8_t ui8SlaveAddr;
    uint8_t ui8Data;
    uint8_t ui8State;
    uint8_t ui8CurrentBit;
    uint8_t ui8IntMask;
    uint8_t ui8IntStatus;
}
tSoftI2C
```

**Members:**

***pfIntCallback*** The address of the callback function that is called to simulate the interrupts that would be produced by a hardware I2C implementation. This address can be set via a direct structure access or using the `SoftI2CCallbackSet` function.

***ui32SCLGPIO*** The address of the GPIO pin to be used for the SCL signal. This member can be set via a direct structure access or using the `SoftI2CSCLGPIOSet` function.

***ui32SDAGPIO*** The address of the GPIO pin to be used for the SDA signal. This member can be set via a direct structure access or using the `SoftI2CSDAGPIOSet` function.

***ui8Flags*** The flags that control the operation of the SoftI2C module. This member should not be accessed or modified by the application.

***ui8SlaveAddr*** The slave address that is currently being accessed. This member should not be accessed or modified by the application.

***ui8Data*** The data that is currently being transmitted or received. This member should not be accessed or modified by the application.

***ui8State*** The current state of the SoftI2C state machine. This member should not be accessed or modified by the application.

***ui8CurrentBit*** The number of bits that have been transmitted and received in the current frame. This member should not be accessed or modified by the application.

***ui8IntMask*** The set of virtual interrupts that should be sent to the callback function. This member should not be accessed or modified by the application.

***ui8IntStatus*** The set of virtual interrupts that are currently asserted. This member should not be accessed or modified by the application.

**Description:**

This structure contains the state of a single instance of a SoftI2C module.

## 15.2.2 Function Documentation

### 15.2.2.1 SoftI2CBusy

Indicates whether or not the SoftI2C module is busy.

**Prototype:**

```
bool  
SoftI2CBusy(tSoftI2C *psI2C)
```

**Parameters:**

***psI2C*** specifies the SoftI2C data structure.

**Description:**

This function returns an indication of whether or not the SoftI2C module is busy transmitting or receiving data.

**Returns:**

Returns **true** if the SoftI2C module is busy; otherwise, returns **false**.

### 15.2.2.2 SoftI2CCallbackSet

Sets the callback used by the SoftI2C module.

**Prototype:**

```
void  
SoftI2CCallbackSet(tSoftI2C *psI2C,  
                  void (*pfnCallback)(void))
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**pfnCallback** is a pointer to the callback function.

**Description:**

This function sets the address of the callback function that is called when there is an “interrupt” produced by the SoftI2C module.

**Returns:**

None.

### 15.2.2.3 SoftI2CControl

Controls the state of the SoftI2C module.

**Prototype:**

```
void  
SoftI2CControl(tSoftI2C *psI2C,  
              uint32_t ui32Cmd)
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**ui32Cmd** command to be issued to the SoftI2C module.

**Description:**

This function is used to control the state of the SoftI2C module send and receive operations. The *ui32Cmd* parameter can be one of the following values:

- **SOFTI2C\_CMD\_SINGLE\_SEND**
- **SOFTI2C\_CMD\_SINGLE\_RECEIVE**
- **SOFTI2C\_CMD\_BURST\_SEND\_START**
- **SOFTI2C\_CMD\_BURST\_SEND\_CONT**
- **SOFTI2C\_CMD\_BURST\_SEND\_FINISH**
- **SOFTI2C\_CMD\_BURST\_SEND\_ERROR\_STOP**
- **SOFTI2C\_CMD\_BURST\_RECEIVE\_START**
- **SOFTI2C\_CMD\_BURST\_RECEIVE\_CONT**
- **SOFTI2C\_CMD\_BURST\_RECEIVE\_FINISH**
- **SOFTI2C\_CMD\_BURST\_RECEIVE\_ERROR\_STOP**

**Returns:**

None.

#### 15.2.2.4 SoftI2CDataGet

Receives a byte that has been sent to the SoftI2C module.

**Prototype:**

```
uint32_t  
SoftI2CDataGet (tSoftI2C *psI2C)
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**Description:**

This function reads a byte of data from the SoftI2C module that was received as a result of an appropriate call to [SoftI2CControl\(\)](#).

**Returns:**

Returns the byte received by the SoftI2C module, cast as an uint32\_t.

#### 15.2.2.5 SoftI2CDataPut

Transmits a byte from the SoftI2C module.

**Prototype:**

```
void  
SoftI2CDataPut (tSoftI2C *psI2C,  
                uint8_t ui8Data)
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**ui8Data** data to be transmitted from the SoftI2C module.

**Description:**

This function places the supplied data into SoftI2C module in preparation for being transmitted via an appropriate call to [SoftI2CControl\(\)](#).

**Returns:**

None.

#### 15.2.2.6 SoftI2CErr

Gets the error status of the SoftI2C module.

**Prototype:**

```
uint32_t  
SoftI2CErr (tSoftI2C *psI2C)
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**Description:**

This function is used to obtain the error status of the SoftI2C module send and receive operations.

**Returns:**

Returns the error status, as one of **SOFTI2C\_ERR\_NONE**, **SOFTI2C\_ERR\_ADDR\_ACK**, or **SOFTI2C\_ERR\_DATA\_ACK**.

### 15.2.2.7 SoftI2CInit

Initializes the SoftI2C module.

**Prototype:**

```
void  
SoftI2CInit (tSoftI2C *psI2C)
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**Description:**

This function initializes operation of the SoftI2C module. After successful initialization of the SoftI2C module, the software I2C bus is in the idle state.

**Returns:**

None.

### 15.2.2.8 SoftI2CIntClear

Clears the SoftI2C “interrupt”.

**Prototype:**

```
void  
SoftI2CIntClear (tSoftI2C *psI2C)
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**Description:**

The SoftI2C “interrupt” source is cleared, so that it no longer asserts. This function must be called in the “interrupt” handler to keep it from being called again immediately on exit.

**Returns:**

None.

### 15.2.2.9 SoftI2CIntDisable

Disables the SoftI2C “interrupt”.

**Prototype:**

```
void  
SoftI2CIntDisable (tSoftI2C *psI2C)
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**Description:**

Disables the SoftI2C “interrupt” source.

**Returns:**

None.

### 15.2.2.10 SoftI2CIntEnable

Enables the SoftI2C “interrupt”.

**Prototype:**

```
void  
SoftI2CIntEnable (tSoftI2C *psI2C)
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**Description:**

Enables the SoftI2C “interrupt” source.

**Returns:**

None.

### 15.2.2.11 SoftI2CIntStatus

Gets the current SoftI2C “interrupt” status.

**Prototype:**

```
bool  
SoftI2CIntStatus (tSoftI2C *psI2C,  
                 bool bMasked)
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**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 SoftI2C 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.

### 15.2.2.12 SoftI2CSCLogPIOSet

Sets the GPIO pin to be used as the SoftI2C SCL signal.

**Prototype:**

```
void  
SoftI2CSCLGPIOSet (tSoftI2C *psI2C,  
                   uint32_t ui32Base,  
                   uint8_t ui8Pin)
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**ui32Base** is the base address of the GPIO module.

**ui8Pin** is the bit-packed representation of the pin to use.

**Description:**

This function sets the GPIO pin that is used for the SoftI2C SCL signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None.

### 15.2.2.13 SoftI2CSDAGPIOSet

Sets the GPIO pin to be used as the SoftI2C SDA signal.

**Prototype:**

```
void  
SoftI2CSDAGPIOSet (tSoftI2C *psI2C,  
                   uint32_t ui32Base,  
                   uint8_t ui8Pin)
```

**Parameters:**

**psI2C** specifies the SoftI2C data structure.

**ui32Base** is the base address of the GPIO module.

**ui8Pin** is the bit-packed representation of the pin to use.

**Description:**

This function sets the GPIO pin that is used for the SoftI2C SDA signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None.

### 15.2.2.14 SoftI2CSlaveAddrSet

Sets the address that the SoftI2C module places on the bus.

**Prototype:**

```
void  
SoftI2CSlaveAddrSet (tSoftI2C *psI2C,  
                    uint8_t ui8SlaveAddr,  
                    bool bReceive)
```



**Parameters:**

***psI2C*** specifies the SoftI2C data structure.

***ui8SlaveAddr*** 7-bit slave address

***bReceive*** flag indicating the type of communication with the slave.

**Description:**

This function sets the address that the SoftI2C module places on the bus when initiating a transaction. When the *bReceive* parameter is set to **true**, the address indicates that the SoftI2C module is initiating a read from the slave; otherwise the address indicates that the SoftI2C module is initiating a write to the slave.

**Returns:**

None.

### 15.2.2.15 SoftI2CTimerTick

Performs the periodic update of the SoftI2C module.

**Prototype:**

```
void  
SoftI2CTimerTick(tSoftI2C *psI2C)
```

**Parameters:**

***psI2C*** specifies the SoftI2C data structure.

**Description:**

This function performs the periodic, time-based updates to the SoftI2C module. The transmission and reception of data over the SoftI2C link is performed by the state machine in this function.

This function must be called at four times the desired SoftI2C clock rate. For example, to run the SoftI2C clock at 10 KHz, this function must be called at a 40 KHz rate.

**Returns:**

None.

## 15.3 Programming Example

The following example shows how to configure the software I2C module and transmit some data to an external peripheral. This example uses Timer 0 as the timing source.

```
//  
// The instance data for the software I2C.  
//  
tSoftI2C g_sI2C;  
  
//  
// The timer tick function.  
//  
void  
Timer0AIntHandler(void)  
{
```

```
//
// Clear the timer interrupt.
//
TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

//
// Call the software I2C timer tick function.
//
SoftI2CTimerTick(&g_sI2C);
}

//
// The callback function for the software I2C. This function is equivalent
// to the interrupt handler for a hardware I2C.
//
void
I2CCallback(void)
{
    //
    // Clear the interrupt.
    //
    SoftI2CIntClear(&g_sI2C);

    //
    // Handle the interrupt.
    //
    ...
}

//
// Setup the software I2C and send some data.
//
void
TestSoftI2C(void)
{
    //
    // Clear the software I2C instance data.
    //
    memset(&g_sI2C, 0, sizeof(g_sI2C));

    //
    // Set the callback function used for this software I2C.
    //
    SoftI2CCallbackSet(&g_sI2C, I2CCallback);

    //
    // Configure the pins used for the software I2C. This example uses
    // pins PD0 and PE1.
    //
    SoftI2CSCLGPIOSet(&g_sI2C, GPIO_PORTD_BASE, GPIO_PIN_0);
    SoftI2CSDAGPIOSet(&g_sI2C, GPIO_PORTE_BASE, GPIO_PIN_1);

    //
    // Enable the GPIO modules that contains the GPIO pins to be used by
    // the software I2C.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

    //
    // Initialize the software I2C module.
    //
    SoftI2CInit(&g_sI2C);

    //
    // Configure the timer used to generate the timing for the software
```

```
// I2C. The interface will be run at 10 KHz, requiring a timer tick
// at 40 KHz.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / 40000);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
IntEnable(INT_TIMER0A);
TimerEnable(TIMER0_BASE, TIMER_A);

//
// Enable the software I2C interrupt.
//
SoftI2CIntEnable(&g_sI2C);

//
// Send a single byte to the slave device.
//
SoftI2CSlaveAddrSet(&g_sI2C, 0x55, 0);
SoftI2CDataPut(&g_sI2C, 0xaa);
SoftI2CControl(&g_sI2C, SOFTI2C_CMD_SINGLE_SEND);

//
// Wait until the software I2C is idle. The completion interrupt will
// be sent to the callback function prior to exiting this loop.
//
while(SoftI2CBusy(&g_sI2C))
{
}
```

As a comparison, the following is the equivalent code using the hardware I2C module and the Tiva Peripheral Driver Library.

```
//
// The interrupt handler for the hardware I2C.
//
void
I2C0IntHandler(void)
{
    //
    // Clear the asserted interrupt sources.
    //
    I2CMasterIntClear(I2C0_MASTER_BASE);

    //
    // Handle the interrupt.
    //
    ...
}

//
// Setup the hardware I2C and send some data.
//
void
TestI2C(void)
{
    //
    // Enable the GPIO module that contains the GPIO pins to be used by
    // the I2C, as well as the I2C module.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);

    //
```

```
// Configure the GPIO pins for use by the I2C module.
//
GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_2 | GPIO_PIN_3);

//
// Initialize the hardware I2C module.
//
I2CMasterInitExpClk(I2C0_MASTER_BASE, SysCtlClockGet(), false);

//
// Enable the hardware I2C.
//
I2CMasterEnable(I2C0_MASTER_BASE);

//
// Enable the interrupt in the hardware I2C.
//
I2CMasterIntEnable(I2C0_MASTER_BASE);
IntEnable(INT_I2C0);

//
// Write some data into the hardware I2C transmit FIFO.
//
I2CMasterSlaveAddrSet(I2C0_MASTER_BASE, 0x55, 0);
I2CMasterDataPut(I2C0_MASTER_BASE, 0xaa);
I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_SINGLE_SEND);

//
// Wait until the hardware I2C is idle. The interrupt will be sent to
// the interrupt handler prior to exiting this loop.
//
while(I2CBusy(I2C0_MASTER_BASE))
{
}
```

## 16 Software SSI Module

Introduction .....	109
API Functions .....	110
Programming Example .....	122

### 16.1 Introduction

The software SSI module uses a timer and a few GPIO pins to create a slow-speed software SSI peripheral. Multiple software SSI modules can be in use simultaneously, limited only by the availability of resources (RAM for the per-instance data structure, pins for the interface, timers if unique clock rates are required, and processor cycles to execute the code). The software SSI module supports the Motorola® SPI™ formats with 4 to 16 data bits. A callback mechanism is used to simulate the interrupts that would be provided by a hardware SSI module.

The API for the software SSI module has been constructed to be as close as possible to the API provided in the Tiva Peripheral Driver Library for the hardware SSI module. The two notable difference are the function prefix being “SoftSSI” instead of “SSI”, and the first argument of each API is a pointer to the `tSoftSSI` data structure instead of the base address of the hardware module.

Timing for the software SSI module is provided by the application. The `SoftSSITimerTick()` function must be called on a periodic basis to provide the timing for the software SSI module. The timer tick function must be called at twice the desired SSI clock rate; for example, to operate the software SSI interface at 10 KHz, the tick function must be called at a 20 KHz rate. By having the application providing the timing, the timer resource to be used is flexible and multiple software SSI modules can be driven from a single timer resource. Alternatively, if the software SSI module is only needed for brief periods of time and processor usage is not a concern, the timer tick function can simply be called in a loop until the entire SSI transaction has completed (maximizing both SSI clock speed and processor usage, but not requiring a timer).

The software SSI module requires as few as two and as many as four GPIO pins. The following table shows the possible pin usages for the software SSI module:

Fss	Clk	Tx	Rx	Pins	Description
	yes	yes		2	transmit only
yes	yes	yes		3	
	yes		yes	2	receive only
yes	yes		yes	3	
	yes	yes	yes	3	transmit and receive
yes	yes	yes	yes	4	

For the cases where Fss is not used, it is up to the application to control that signal (either via a separately-controlled GPIO, or by being tied to ground in the hardware).

The per-instance data structure is approximately 52 bytes in length (the actual length will depend upon how the structure is packed by the compiler in use).

As a point of reference, the following are some rough measurements of the processor usage of the software SSI module at various SSI clock speeds with the processor running at 50 MHz. Actual processor usage may vary, depending upon how the application uses the software SSI module, processor clock speed, interrupt priority, and compiler in use.

SSI Clock	% Of Processor	Million Cycles Per Second
10 KHz	5.26	2.63
20 KHz	10.48	5.24
30 KHz	15.68	7.84
40 KHz	20.90	10.45
50 KHz	26.10	13.05
60 KHz	31.38	15.69
70 KHz	36.54	18.27
80 KHz	41.79	20.89
90 KHz	47.06	23.53
100 KHz	52.17	26.08

This module is contained in `utils/softssi.c`, with `utils/softssi.h` containing the API declarations for use by applications.

## 16.2 API Functions

### Data Structures

- `tSoftSSI`

### Functions

- `bool SoftSSIBusy (tSoftSSI *psSSI)`
- `void SoftSSICallbackSet (tSoftSSI *psSSI, void (*pfnCallback)(void))`
- `void SoftSSIClkGPIOSet (tSoftSSI *psSSI, uint32_t ui32Base, uint8_t ui8Pin)`
- `void SoftSSIConfigSet (tSoftSSI *psSSI, uint8_t ui8Protocol, uint8_t ui8Bits)`
- `bool SoftSSIDataAvail (tSoftSSI *psSSI)`
- `void SoftSSIDataGet (tSoftSSI *psSSI, uint32_t *pui32Data)`
- `int32_t SoftSSIDataGetNonBlocking (tSoftSSI *psSSI, uint32_t *pui32Data)`
- `void SoftSSIDataPut (tSoftSSI *psSSI, uint32_t ui32Data)`
- `int32_t SoftSSIDataPutNonBlocking (tSoftSSI *psSSI, uint32_t ui32Data)`
- `void SoftSSIDisable (tSoftSSI *psSSI)`
- `void SoftSSIEnable (tSoftSSI *psSSI)`
- `void SoftSSIFssGPIOSet (tSoftSSI *psSSI, uint32_t ui32Base, uint8_t ui8Pin)`
- `void SoftSSIIntClear (tSoftSSI *psSSI, uint32_t ui32IntFlags)`
- `void SoftSSIIntDisable (tSoftSSI *psSSI, uint32_t ui32IntFlags)`
- `void SoftSSIIntEnable (tSoftSSI *psSSI, uint32_t ui32IntFlags)`
- `uint32_t SoftSSIIntStatus (tSoftSSI *psSSI, bool bMasked)`
- `void SoftSSIRxBufferSet (tSoftSSI *psSSI, uint16_t *pui16RxBuffer, uint16_t ui16Len)`
- `void SoftSSIRxGPIOSet (tSoftSSI *psSSI, uint32_t ui32Base, uint8_t ui8Pin)`
- `bool SoftSSISpaceAvail (tSoftSSI *psSSI)`
- `void SoftSSITimerTick (tSoftSSI *psSSI)`
- `void SoftSSITxBufferSet (tSoftSSI *psSSI, uint16_t *pui16TxBuffer, uint16_t ui16Len)`
- `void SoftSSITxGPIOSet (tSoftSSI *psSSI, uint32_t ui32Base, uint8_t ui8Pin)`

## 16.2.1 Data Structure Documentation

### 16.2.1.1 tSoftSSI

**Definition:**

```
typedef struct
{
    void (*pfnIntCallback) (void);
    uint32_t ui32FssGPIO;
    uint32_t ui32ClkGPIO;
    uint32_t ui32TxGPIO;
    uint32_t ui32RxGPIO;
    uint16_t *pui16TxBuffer;
    uint16_t *pui16RxBuffer;
    uint16_t ui16TxBufferLen;
    uint16_t ui16TxBufferRead;
    uint16_t ui16TxBufferWrite;
    uint16_t ui16RxBufferLen;
    uint16_t ui16RxBufferRead;
    uint16_t ui16RxBufferWrite;
    uint16_t ui16TxData;
    uint16_t ui16RxData;
    uint8_t ui8Flags;
    uint8_t ui8Bits;
    uint8_t ui8State;
    uint8_t ui8CurrentBit;
    uint8_t ui8IntMask;
    uint8_t ui8IntStatus;
    uint8_t ui8IdleCount;
}
tSoftSSI
```

**Members:**

***pfnIntCallback*** The address of the callback function that is called to simulate the interrupts that would be produced by a hardware SSI implementation. This address can be set via a direct structure access or using the SoftSSICallbackSet function.

***ui32FssGPIO*** The address of the GPIO pin to be used for the Fss signal. If this member is zero, the Fss signal is not generated. This member can be set via a direct structure access or using the SoftSSIFssGPIOSet function.

***ui32ClkGPIO*** The address of the GPIO pin to be used for the Clk signal. This member can be set via a direct structure access or using the SoftSSIClkGPIOSet function.

***ui32TxGPIO*** The address of the GPIO pin to be used for the Tx signal. This member can be set via a direct structure access or using the SoftSSITxGPIOSet function.

***ui32RxGPIO*** The address of the GPIO pin to be used for the Rx signal. If this member is zero, the Rx signal is not read. This member can be set via a direct structure access or using the SoftSSIRxGPIOSet function.

***pui16TxBuffer*** The address of the data buffer used for the transmit FIFO. This member can be set via a direct structure access or using the SoftSSITxBufferSet function.

***pui16RxBuffer*** The address of the data buffer used for the receive FIFO. This member can be set via a direct structure access or using the SoftSSIRxBufferSet function.

***ui16TxBufferLen*** The length of the transmit FIFO. This member can be set via a direct structure access or using the SoftSSITxBufferSet function.

- ui16TxBufferRead** The index into the transmit FIFO of the next word to be transmitted. This member should be initialized to zero, but should not be accessed or modified by the application.
- ui16TxBufferWrite** The index into the transmit FIFO of the next location to store data into the FIFO. This member should be initialized to zero, but should not be accessed or modified by the application.
- ui16RxBufferLen** The length of the receive FIFO. This member can be set via a direct structure access or using the `SoftSSIRxBufferSet` function.
- ui16RxBufferRead** The index into the receive FIFO of the next word to be read from the FIFO. This member should be initialized to zero, but should not be accessed or modified by the application.
- ui16RxBufferWrite** The index into the receive FIFO of the location to store the next word received. This member should be initialized to zero, but should not be accessed or modified by the application.
- ui16TxData** The word that is currently being transmitted. This member should not be accessed or modified by the application.
- ui16RxData** The word that is currently being received. This member should not be accessed or modified by the application.
- ui8Flags** The flags that control the operation of the SoftSSI module. This member should not be accessed or modified by the application.
- ui8Bits** The number of data bits in each SoftSSI frame, which also specifies the width of each data item in the transmit and receive FIFOs. This member can be set via a direct structure access or using the `SoftSSIConfigSet` function.
- ui8State** The current state of the SoftSSI state machine. This member should not be accessed or modified by the application.
- ui8CurrentBit** The number of bits that have been transmitted and received in the current frame. This member should not be accessed or modified by the application.
- ui8IntMask** The set of virtual interrupts that should be sent to the callback function. This member should not be accessed or modified by the application.
- ui8IntStatus** The set of virtual interrupts that are currently asserted. This member should not be accessed or modified by the application.
- ui8IdleCount** The number of tick counts that the SoftSSI module has been idle with data stored in the receive FIFO, which is used to generate the receive timeout interrupt. This member should not be accessed or modified by the application.

**Description:**

This structure contains the state of a single instance of a SoftSSI module.

## 16.2.2 Function Documentation

### 16.2.2.1 SoftSSIBusy

Determines whether the SoftSSI transmitter is busy or not.

**Prototype:**

```
bool  
SoftSSIBusy(tSoftSSI *psSSI)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.



**Description:**

Allows the caller to determine whether all transmitted bytes have cleared the transmitter. If **false** is returned, then the transmit FIFO is empty and all bits of the last transmitted word have left the shift register.

**Returns:**

Returns **true** if the SoftSSI is transmitting or **false** if all transmissions are complete.

### 16.2.2.2 SoftSSICallbackSet

Sets the callback used by the SoftSSI module.

**Prototype:**

```
void  
SoftSSICallbackSet (tSoftSSI *psSSI,  
                   void (*pfnCallback) (void))
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**pfnCallback** is a pointer to the callback function.

**Description:**

This function sets the address of the callback function that is called when there is an “interrupt” produced by the SoftSSI module.

**Returns:**

None.

### 16.2.2.3 SoftSSIClkGPIOSet

Sets the GPIO pin to be used as the SoftSSI Clk signal.

**Prototype:**

```
void  
SoftSSIClkGPIOSet (tSoftSSI *psSSI,  
                  uint32_t ui32Base,  
                  uint8_t ui8Pin)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**ui32Base** is the base address of the GPIO module.

**ui8Pin** is the bit-packed representation of the pin to use.

**Description:**

This function sets the GPIO pin that is used for the SoftSSI Clk signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None.

#### 16.2.2.4 SoftSSISetConfig

Sets the configuration of a SoftSSI module.

**Prototype:**

```
void  
SoftSSISetConfig(tSoftSSI *psSSI,  
                uint8_t ui8Protocol,  
                uint8_t ui8Bits)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**ui8Protocol** specifies the data transfer protocol.

**ui8Bits** specifies the number of bits transferred per frame.

**Description:**

This function configures the data format of a SoftSSI module. The *ui8Protocol* parameter can be one of the following values: **SOFTSSI\_FRF\_MOTO\_MODE\_0**, **SOFTSSI\_FRF\_MOTO\_MODE\_1**, **SOFTSSI\_FRF\_MOTO\_MODE\_2**, or **SOFTSSI\_FRF\_MOTO\_MODE\_3**. These frame formats imply the following polarity and phase configurations:

Polarity	Phase	Mode
0	0	SOFTSSI_FRF_MOTO_MODE_0
0	1	SOFTSSI_FRF_MOTO_MODE_1
1	0	SOFTSSI_FRF_MOTO_MODE_2
1	1	SOFTSSI_FRF_MOTO_MODE_3

The *ui8Bits* parameter defines the width of the data transfers, and can be a value between 4 and 16, inclusive.

**Returns:**

None.

#### 16.2.2.5 SoftSSIDataAvail

Determines if there is any data in the receive FIFO.

**Prototype:**

```
bool  
SoftSSIDataAvail(tSoftSSI *psSSI)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**Description:**

This function determines if there is any data available to be read from the receive FIFO.

**Returns:**

Returns **true** if there is data in the receive FIFO or **false** if there is no data in the receive FIFO.

### 16.2.2.6 SoftSSIDataGet

Gets a data element from the SoftSSI receive FIFO.

**Prototype:**

```
void  
SoftSSIDataGet (tSoftSSI *psSSI,  
                uint32_t *pui32Data)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**pui32Data** is a pointer to a storage location for data that was received over the SoftSSI interface.

**Description:**

This function gets received data from the receive FIFO of the specified SoftSSI module and places that data into the location specified by the *pui32Data* parameter.

**Note:**

Only the lower N bits of the value written to *pui32Data* contain valid data, where N is the data width as configured by [SoftSSISetConfig\(\)](#). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *pui32Data* contain valid data.

**Returns:**

None.

### 16.2.2.7 SoftSSIDataGetNonBlocking

Gets a data element from the SoftSSI receive FIFO.

**Prototype:**

```
int32_t  
SoftSSIDataGetNonBlocking (tSoftSSI *psSSI,  
                           uint32_t *pui32Data)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**pui32Data** is a pointer to a storage location for data that was received over the SoftSSI interface.

**Description:**

This function gets received data from the receive FIFO of the specified SoftSSI module and places that data into the location specified by the *pui32Data* parameter. If there is no data in the FIFO, then this function returns a zero.

**Note:**

Only the lower N bits of the value written to *pui32Data* contain valid data, where N is the data width as configured by [SoftSSISetConfig\(\)](#). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *pui32Data* contain valid data.

**Returns:**

Returns the number of elements read from the SoftSSI receive FIFO.

### 16.2.2.8 SoftSSIDataPut

Puts a data element into the SoftSSI transmit FIFO.

**Prototype:**

```
void  
SoftSSIDataPut (tSoftSSI *psSSI,  
                uint32_t ui32Data)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**ui32Data** is the data to be transmitted over the SoftSSI interface.

**Description:**

This function places the supplied data into the transmit FIFO of the specified SoftSSI module.

**Note:**

The upper 32 - N bits of the *ui32Data* are discarded, where N is the data width as configured by [SoftSSIConfigSet\(\)](#). For example, if the interface is configured for 8-bit data width, the upper 24 bits of *ui32Data* are discarded.

**Returns:**

None.

### 16.2.2.9 SoftSSIDataPutNonBlocking

Puts a data element into the SoftSSI transmit FIFO.

**Prototype:**

```
int32_t  
SoftSSIDataPutNonBlocking (tSoftSSI *psSSI,  
                            uint32_t ui32Data)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**ui32Data** is the data to be transmitted over the SoftSSI interface.

**Description:**

This function places the supplied data into the transmit FIFO of the specified SoftSSI module. If there is no space in the FIFO, then this function returns a zero.

**Note:**

The upper 32 - N bits of the *ui32Data* are discarded, where N is the data width as configured by [SoftSSIConfigSet\(\)](#). For example, if the interface is configured for 8-bit data width, the upper 24 bits of *ui32Data* are discarded.

**Returns:**

Returns the number of elements written to the SSI transmit FIFO.

### 16.2.2.10 SoftSSIDisable

Disables the SoftSSI module.

**Prototype:**

```
void  
SoftSSIDisable (tSoftSSI *psSSI)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**Description:**

This function disables operation of the SoftSSI module. If a data transfer is in progress, it is finished before the module is fully disabled.

**Returns:**

None.

### 16.2.2.11 SoftSSIEnable

Enables the SoftSSI module.

**Prototype:**

```
void  
SoftSSIEnable (tSoftSSI *psSSI)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**Description:**

This function enables operation of the SoftSSI module. The SoftSSI module must be configured before it is enabled.

**Returns:**

None.

### 16.2.2.12 SoftSSIFssGPIOSet

Sets the GPIO pin to be used as the SoftSSI Fss signal.

**Prototype:**

```
void  
SoftSSIFssGPIOSet (tSoftSSI *psSSI,  
                   uint32_t ui32Base,  
                   uint8_t ui8Pin)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**ui32Base** is the base address of the GPIO module.

**ui8Pin** is the bit-packed representation of the pin to use.

**Description:**

This function sets the GPIO pin that is used for the SoftSSI Fss signal. If there is not a GPIO pin allocated for Fss, the SoftSSI module does not assert/deassert the Fss signal, leaving it to the application either to do manually or to not do at all if the slave device has Fss tied to ground.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None.

### 16.2.2.13 SoftSSIIntClear

Clears SoftSSI “interrupt” sources.

**Prototype:**

```
void  
SoftSSIIntClear(tSoftSSI *psSSI,  
                uint32_t ui32IntFlags)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**ui32IntFlags** is a bit mask of the “interrupt” sources to be cleared.

**Description:**

The specified SoftSSI “interrupt” sources are cleared so that they no longer assert. This function must be called in the “interrupt” handler to keep the “interrupt” from being recognized again immediately upon exit. The *ui32IntFlags* parameter is the logical OR of any of the **SOFTSSI\_TXEOT**, **SOFTSSI\_RXTO**, and **SOFTSSI\_RXOR** values.

**Returns:**

None.

### 16.2.2.14 SoftSSIIntDisable

Disables individual SoftSSI “interrupt” sources.

**Prototype:**

```
void  
SoftSSIIntDisable(tSoftSSI *psSSI,  
                  uint32_t ui32IntFlags)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**ui32IntFlags** is a bit mask of the “interrupt” sources to be disabled.

**Description:**

Disables the indicated SoftSSI “interrupt” sources. The *ui32IntFlags* parameter can be any of the **SOFTSSI\_TXEOT**, **SOFTSSI\_TXFF**, **SOFTSSI\_RXFF**, **SOFTSSI\_RXTO**, or **SOFTSSI\_RXOR** values.

**Returns:**

None.

### 16.2.2.15 SoftSSIIntEnable

Enables individual SoftSSI “interrupt” sources.

**Prototype:**

```
void  
SoftSSIIntEnable (tSoftSSI *psSSI,  
                 uint32_t ui32IntFlags)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**ui32IntFlags** is a bit mask of the “interrupt” sources to be enabled.

**Description:**

Enables the indicated SoftSSI “interrupt” sources. Only the sources that are enabled can be reflected to the callback function; disabled sources do not result in a callback. The *ui32IntFlags* parameter can be any of the **SOFTSSI\_TXEOT**, **SOFTSSI\_TXFF**, **SOFTSSI\_RXFF**, **SOFTSSI\_RXT0**, or **SOFTSSI\_RXOR** values.

**Returns:**

None.

### 16.2.2.16 SoftSSIIntStatus

Gets the current “interrupt” status.

**Prototype:**

```
uint32_t  
SoftSSIIntStatus (tSoftSSI *psSSI,  
                 bool bMasked)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**bMasked** is **false** if the raw “interrupt” status is required or **true** if the masked “interrupt” status is required.

**Description:**

This function returns the “interrupt” status for the SoftSSI module. Either the raw “interrupt” status or the status of “interrupts” that are allowed to reflect to the callback can be returned.

**Returns:**

The current “interrupt” status, enumerated as a bit field of **SOFTSSI\_TXEOT**, **SOFTSSI\_TXFF**, **SOFTSSI\_RXFF**, **SOFTSSI\_RXT0**, and **SOFTSSI\_RXOR**.

### 16.2.2.17 SoftSSIRxBufferSet

Sets the receive FIFO buffer for a SoftSSI module.

**Prototype:**

```
void  
SoftSSIRxBufferSet (tSoftSSI *psSSI,  
                    uint16_t *pui16RxBuffer,  
                    uint16_t ui16Len)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**pui16RxBuffer** is the address of the receive FIFO buffer.

**ui16Len** is the size, in 16-bit half-words, of the receive FIFO buffer.

**Description:**

This function sets the address and size of the receive FIFO buffer and also resets the read and write pointers, marking the receive FIFO as empty. When the buffer pointer and length are configured as zero, all data received from the slave device is discarded. This capability is useful when there is no GPIO pin allocated for the Rx signal.

**Returns:**

None.

### 16.2.2.18 SoftSSIRxGPIOSet

Sets the GPIO pin to be used as the SoftSSI Rx signal.

**Prototype:**

```
void  
SoftSSIRxGPIOSet (tSoftSSI *psSSI,  
                  uint32_t ui32Base,  
                  uint8_t ui8Pin)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**ui32Base** is the base address of the GPIO module.

**ui8Pin** is the bit-packed representation of the pin to use.

**Description:**

This function sets the GPIO pin that is used for the SoftSSI Rx signal. If there is not a GPIO pin allocated for Rx, the SoftSSI module does not read data from the slave device.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None.



### 16.2.2.19 SoftSSISpaceAvail

Determines if there is any space in the transmit FIFO.

**Prototype:**

```
bool  
SoftSSISpaceAvail (tSoftSSI *psSSI)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**Description:**

This function determines if there is space available in the transmit FIFO.

**Returns:**

Returns **true** if there is space available in the transmit FIFO or **false** if there is no space available in the transmit FIFO.

### 16.2.2.20 SoftSSITimerTick

Performs the periodic update of the SoftSSI module.

**Prototype:**

```
void  
SoftSSITimerTick (tSoftSSI *psSSI)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

**Description:**

This function performs the periodic, time-based updates to the SoftSSI module. The transmission and reception of data over the SoftSSI link is performed by the state machine in this function.

This function must be called at twice the desired SoftSSI clock rate. For example, to run the SoftSSI clock at 10 KHz, this function must be called at a 20 KHz rate.

**Returns:**

None.

### 16.2.2.21 SoftSSITxBufferSet

Sets the transmit FIFO buffer for a SoftSSI module.

**Prototype:**

```
void  
SoftSSITxBufferSet (tSoftSSI *psSSI,  
                    uint16_t *puil6TxBuffer,  
                    uint16_t uil6Len)
```

**Parameters:**

**psSSI** specifies the SoftSSI data structure.

***psTxBuffer*** is the address of the transmit FIFO buffer.

***ui16Len*** is the size, in 16-bit half-words, of the transmit FIFO buffer.

**Description:**

This function sets the address and size of the transmit FIFO buffer and also resets the read and write pointers, marking the transmit FIFO as empty.

**Returns:**

None.

### 16.2.2.22 SoftSSITxGPIOSet

Sets the GPIO pin to be used as the SoftSSI Tx signal.

**Prototype:**

```
void  
SoftSSITxGPIOSet (tSoftSSI *psSSI,  
                  uint32_t ui32Base,  
                  uint8_t ui8Pin)
```

**Parameters:**

***psSSI*** specifies the SoftSSI data structure.

***ui32Base*** is the base address of the GPIO module.

***ui8Pin*** is the bit-packed representation of the pin to use.

**Description:**

This function sets the GPIO pin that is used for the SoftSSI Tx signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None.

## 16.3 Programming Example

The following example shows how to configure the software SSI module and transmit some data to an external peripheral. This example uses Timer 0 as the timing source.

```
//  
// The instance data for the software SSI.  
//  
tSoftSSI g_sSSI;  
  
//  
// The buffer used to hold the transmit data.  
//  
unsigned short g_pusTxBuffer[8];  
  
//  
// The timer tick function.  
//
```

```
void
Timer0AIntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Call the software SSI timer tick function.
    //
    SoftSSITimerTick(&g_sSSI);
}

//
// The callback function for the software SSI. This function is equivalent
// to the interrupt handler for a hardware SSI.
//
void
SSICallback(void)
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = SoftSSIIntStatus(&g_sSSI, true);

    //
    // Clear the asserted interrupt sources.
    //
    SoftSSIIntClear(&g_sSSI, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the software SSI and send some data.
//
void
TestSoftSSI(void)
{
    //
    // Clear the software SSI instance data.
    //
    memset(&g_sSSI, 0, sizeof(g_sSSI));

    //
    // Set the callback function used for this software SSI.
    //
    SoftSSICallbackSet(&g_sSSI, SSICallback);

    //
    // Configure the pins used for the software SSI. This example uses
    // pins PD0, PE1, and PF2.
    //
    SoftSSIFssGPIOSet(&g_sSSI, GPIO_PORTD_BASE, GPIO_PIN_0);
    SoftSSIClkGPIOSet(&g_sSSI, GPIO_PORTE_BASE, GPIO_PIN_1);
    SoftSSITxGPIOSet(&g_sSSI, GPIO_PORTF_BASE, GPIO_PIN_2);

    //
    // Configure the data buffer used as the transmit FIFO.
    //
}
```

```
SoftSSITxBufferSet(&g_sSSI, g_pusTxBuffer, 8);

//
// Enable the GPIO modules that contains the GPIO pins to be used by
// the software SSI.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

//
// Initialize the software SSI module, using mode 3 and 8 data bits.
//
SoftSSIConfigSet(&g_sSSI, SOFTSSI_FRF_MOTO_MODE_3, 8);

//
// Enable the software SSI.
//
SoftSSIEnable(&g_sSSI);

//
// Configure the timer used to generate the timing for the software
// SSI. The interface will be run at 10 KHz, requiring a timer tick
// at 20 KHz.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / 20000);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
IntEnable(INT_TIMER0A);
TimerEnable(TIMER0_BASE, TIMER_A);

//
// Enable the transmit FIFO half full interrupt in the software SSI.
//
SoftSSIIntEnable(&g_sSSI, SOFTSSI_TXFF);

//
// Write some data into the software SSI transmit FIFO.
//
SoftSSIDataPut(&g_sSSI, 0x55);
SoftSSIDataPut(&g_sSSI, 0xaa);
SoftSSIDataPut(&g_sSSI, 0x55);
SoftSSIDataPut(&g_sSSI, 0xaa);
SoftSSIDataPut(&g_sSSI, 0x55);
SoftSSIDataPut(&g_sSSI, 0xaa);

//
// Wait until the software SSI is idle. The transmit FIFO half full
// interrupt will be sent to the callback function prior to exiting
// this loop.
//
while(SoftSSIBusy(&g_sSSI))
{
}
}
```

As a comparison, the following is the equivalent code using the hardware SSI module and the Tiva Peripheral Driver Library.

```
//
// The interrupt handler for the hardware SSI.
//
void
SSI0IntHandler(void)
```

```
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = SSIIntStatus(SSIO_BASE, true);

    //
    // Clear the asserted interrupt sources.
    //
    SSIIntClear(SSIO_BASE, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the hardware SSI and send some data.
//
void
TestSSI(void)
{
    //
    // Enable the GPIO module that contains the GPIO pins to be used by
    // the SSI, as well as the SSI module.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);

    //
    // Configure the GPIO pins for use by the SSI module.
    //
    GPIOPinTypeSSI(GPIO_PORTA_BASE, (GPIO_PIN_2 | GPIO_PIN_3 |
                                     GPIO_PIN_4 | GPIO_PIN_5));

    //
    // Initialize the hardware SSI module, using mode 3 and 8 data bits.
    //
    SSIConfigSetExpClk(SSIO_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_3,
                      SSI_MODE_MASTER, 10000, 8);

    //
    // Enable the hardware SSI.
    //
    SSIEnable(SSIO_BASE);

    //
    // Enable the transmit FIFO half full interrupt in the hardware SSI.
    //
    SSIIntEnable(SSIO_BASE, SSI_TXFF);
    IntEnable(INT_SSI0);

    //
    // Write some data into the hardware SSI transmit FIFO.
    //
    SSIDataPut(SSIO_BASE, 0x55);
    SSIDataPut(SSIO_BASE, 0xaa);
    SSIDataPut(SSIO_BASE, 0x55);
    SSIDataPut(SSIO_BASE, 0xaa);
    SSIDataPut(SSIO_BASE, 0x55);
    SSIDataPut(SSIO_BASE, 0xaa);

    //

```

```
// Wait until the hardware SSI is idle. The transmit FIFO half full
// interrupt will be sent to the interrupt handler prior to exiting
// this loop.
//
while(SSIBusy(SSIO_BASE))
{
}
```

## 17 Software UART Module

Introduction .....	127
API Functions .....	128
Programming Example .....	144

### 17.1 Introduction

The software UART module uses two timers and a two GPIO pins to create a software UART peripheral. Multiple software UART modules can be in use simultaneously, limited only by the availability of resources (RAM for the per-instance data structure, pins for the interface, timers, and processor cycles to execute the code). The software UART module supports five through eight data bits, a variety of parity modes (odd, even, one, zero, and none), and one or two stop bits. A callback mechanism is used to simulate the interrupts that would be provided by a hardware UART module.

The API for the software UART module has been constructed to be as close as possible to the API provided in the Tiva Peripheral Driver Library for the hardware UART module. The two notable difference are the function prefix being “SoftUART” instead of “UART”, and the first argument of each API is a pointer to the [tSoftUART](#) data structure instead of the base address of the hardware module.

The software UART transmitter and receiver are handled independently (because of the asynchronous nature of the two). As a result, there are separate timers for each, and if only one is required then the other does not need to be utilized.

Timing for the software UART transmitter is provided by the application. The [SoftUARTTx-TimerTick\(\)](#) function must be called on a periodic basis to provide the timing for the software UART transmitter. The timer tick function must be called at the desired UART baud rate; for example, to operate the software UART transmitter at 38,400 baud, the tick function must be called at a 38,400 Hz rate. Because the application provides the timing, the timer resource can be flexible and multiple software UART transmitters can be driven from a single timer resource.

Timing for the software UART receiver is also provided by the application. Initially, the Rx pin is configured by the software UART module for a GPIO edge interrupt. The GPIO edge interrupt handler must be provided by the application (so that it can be shared with other possible GPIO interrupts on that port). When the interrupt occurs, a timer must be started at the desired baud rate (i.e. for 38,400 baud, it must run at 38,400 Hz) and the [SoftUARTRxTick\(\)](#) function must be called. Then, whenever the timer interrupt occurs, the [SoftUARTRxTick\(\)](#) function must be called. The timer is disabled whenever [SoftUARTRxTick\(\)](#) indicates that it is no longer needed. Because the application provides the timing, the timer resource can be flexible. However, each software UART receiver must have its own timer resource.

The software UART module requires one or two GPIO pins. The following table shows the possible pin usages for the software UART module:

Tx	Rx	Pins	Description
yes		1	transmit only
	yes	1	receive only
yes	yes	2	transmit and receive

The per-instance data structure is approximately 52 bytes in length (the actual length depends on how the structure is packed by the compiler in use).

The following table shows some approximate measurements of the processor usage of the software UART module at various baud rates with the processor running at 50 MHz. Actual processor usage may vary, depending on how the application uses the software UART module, processor clock speed, interrupt priority, and compiler in use.

UART Baud Rate	% Of Processor	Million Cycles Per Second
9600	5.32	2.66
14400	7.99	3.99
19200	10.65	5.32
28800	15.96	7.98
38400	21.28	10.64
57600	32.00	16.00
115200	64.04	32.02

This module is contained in `utils/softuart.c`, with `utils/softuart.h` containing the API declarations for use by applications.

## 17.2 API Functions

### Data Structures

- [tSoftUART](#)

### Functions

- void [SoftUARTBreakCtl](#) ([tSoftUART](#) \*psUART, bool bBreakState)
- bool [SoftUARTBusy](#) ([tSoftUART](#) \*psUART)
- void [SoftUARTCallbackSet](#) ([tSoftUART](#) \*psUART, void (\*pfnCallback)(void))
- int32\_t [SoftUARTCharGet](#) ([tSoftUART](#) \*psUART)
- int32\_t [SoftUARTCharGetNonBlocking](#) ([tSoftUART](#) \*psUART)
- void [SoftUARTCharPut](#) ([tSoftUART](#) \*psUART, uint8\_t ui8Data)
- bool [SoftUARTCharPutNonBlocking](#) ([tSoftUART](#) \*psUART, uint8\_t ui8Data)
- bool [SoftUARTCharsAvail](#) ([tSoftUART](#) \*psUART)
- void [SoftUARTConfigGet](#) ([tSoftUART](#) \*psUART, uint32\_t \*pui32Config)
- void [SoftUARTConfigSet](#) ([tSoftUART](#) \*psUART, uint32\_t ui32Config)
- void [SoftUARTDisable](#) ([tSoftUART](#) \*psUART)
- void [SoftUARTEnable](#) ([tSoftUART](#) \*psUART)
- void [SoftUARTFIFOLevelGet](#) ([tSoftUART](#) \*psUART, uint32\_t \*pui32TxLevel, uint32\_t \*pui32RxLevel)
- void [SoftUARTFIFOLevelSet](#) ([tSoftUART](#) \*psUART, uint32\_t ui32TxLevel, uint32\_t ui32RxLevel)
- void [SoftUARTInit](#) ([tSoftUART](#) \*psUART)
- void [SoftUARTIntClear](#) ([tSoftUART](#) \*psUART, uint32\_t ui32IntFlags)
- void [SoftUARTIntDisable](#) ([tSoftUART](#) \*psUART, uint32\_t ui32IntFlags)
- void [SoftUARTIntEnable](#) ([tSoftUART](#) \*psUART, uint32\_t ui32IntFlags)



- uint32\_t [SoftUARTIntStatus](#) (tSoftUART \*psUART, bool bMasked)
- uint32\_t [SoftUARTParityModeGet](#) (tSoftUART \*psUART)
- void [SoftUARTParityModeSet](#) (tSoftUART \*psUART, uint32\_t ui32Parity)
- void [SoftUARTRxBufferSet](#) (tSoftUART \*psUART, uint16\_t \*pui16RxBuffer, uint16\_t ui16Len)
- void [SoftUARTRxErrorClear](#) (tSoftUART \*psUART)
- uint32\_t [SoftUARTRxErrorGet](#) (tSoftUART \*psUART)
- void [SoftUARTRxGPIOSet](#) (tSoftUART \*psUART, uint32\_t ui32Base, uint8\_t ui8Pin)
- uint32\_t [SoftUARTRxTick](#) (tSoftUART \*psUART, bool bEdgeInt)
- bool [SoftUARTSpaceAvail](#) (tSoftUART \*psUART)
- void [SoftUARTTxBufferSet](#) (tSoftUART \*psUART, uint8\_t \*pui8TxBuffer, uint16\_t ui16Len)
- void [SoftUARTTxGPIOSet](#) (tSoftUART \*psUART, uint32\_t ui32Base, uint8\_t ui8Pin)
- void [SoftUARTTxTimerTick](#) (tSoftUART \*psUART)

## 17.2.1 Data Structure Documentation

### 17.2.1.1 tSoftUART

#### Definition:

```
typedef struct
{
    void (*pfnIntCallback) (void);
    uint32_t ui32TxGPIO;
    uint32_t ui32RxGPIOPort;
    uint8_t *pui8TxBuffer;
    uint16_t *pui16RxBuffer;
    uint16_t ui16TxBufferLen;
    uint16_t ui16TxBufferRead;
    uint16_t ui16TxBufferWrite;
    uint16_t ui16TxBufferLevel;
    uint16_t ui16RxBufferLen;
    uint16_t ui16RxBufferRead;
    uint16_t ui16RxBufferWrite;
    uint16_t ui16RxBufferLevel;
    uint16_t ui16IntStatus;
    uint16_t ui16IntMask;
    uint16_t ui16Config;
    uint8_t ui8Flags;
    uint8_t ui8TxState;
    uint8_t ui8TxNext;
    uint8_t ui8TxData;
    uint8_t ui8RxPin;
    uint8_t ui8RxState;
    uint8_t ui8RxData;
    uint8_t ui8RxFlags;
    uint8_t ui8RxStatus;
}
tSoftUART
```

#### Members:

***pfnIntCallback*** The address of the callback function that is called to simulate the interrupts

that would be produced by a hardware UART implementation. This address can be set via a direct structure access or using the `SoftUARTCallbackSet` function.

***ui32TxGPIO*** The address of the GPIO pin to be used for the Tx signal. This member can be set via a direct structure access or using the `SoftUARTTxGPIOSet` function.

***ui32RxGPIOPort*** The address of the GPIO port to be used for the Rx signal. This member can be set via a direct structure access or using the `SoftUARTRxGPIOSet` function.

***pui8TxBuffer*** The address of the data buffer used for the transmit buffer. This member can be set via a direct structure access or using the `SoftUARTTxBufferSet` function.

***pui16RxBuffer*** The address of the data buffer used for the receive buffer. This member can be set via a direct structure access or using the `SoftUARTRxBufferSet` function.

***ui16TxBufferLen*** The length of the transmit buffer. This member can be set via a direct structure access or using the `SoftUARTTxBufferSet` function.

***ui16TxBufferRead*** The index into the transmit buffer of the next character to be transmitted. This member should not be accessed or modified by the application.

***ui16TxBufferWrite*** The index into the transmit buffer of the next location to store a character into the buffer. This member should not be accessed or modified by the application.

***ui16TxBufferLevel*** The transmit buffer level at which the transmit interrupt is asserted. This member should not be accessed or modified by the application.

***ui16RxBufferLen*** The length of the receive buffer. This member can be set via a direct structure access or using the `SoftUARTRxBufferSet` function.

***ui16RxBufferRead*** The index into the receive buffer of the next character to be read from the buffer. This member should not be accessed or modified by the application.

***ui16RxBufferWrite*** The index into the receive buffer of the location to store the next character received. This member should not be accessed or modified by the application.

***ui16RxBufferLevel*** The receive buffer level at which the receive interrupt is asserted. This member should not be accessed or modified by the application.

***ui16IntStatus*** The set of virtual interrupts that are currently asserted. This member should not be accessed or modified by the application.

***ui16IntMask*** The set of virtual interrupts that should be sent to the callback function. This member should not be accessed or modified by the application.

***ui16Config*** The configuration of the SoftUART module. This member can be set via the `SoftUARTConfigSet` and `SoftUARTFIFOLevelSet` functions.

***ui8Flags*** The flags that control the operation of the SoftUART module. This member should not be accessed or modified by the application.

***ui8TxState*** The current state of the SoftUART transmit state machine. This member should not be accessed or modified by the application.

***ui8TxNext*** The value that is written to the Tx pin at the start of the next transmit timer tick. This member should not be accessed or modified by the application.

***ui8TxData*** The character that is currently being sent via the Tx pin. This member should not be accessed or modified by the application.

***ui8RxPin*** The GPIO pin to be used for the Rx signal. This member can be set via a direct structure access or using the `SoftUARTRxGPIOSet` function.

***ui8RxState*** The current state of the SoftUART receive state machine. This member should not be accessed or modified by the application.

***ui8RxData*** The character that is currently being received via the Rx pin. This member should not be accessed or modified by the application.

***ui8RxFlags*** The flags that indicate any errors that have occurred during the reception of the current character via the Rx pin. This member should not be accessed or modified by the application.

**ui8RxStatus** The receive error status. This member should only be accessed via the SoftUARTRxErrorGet and SoftURATRxErrorClear functions.

**Description:**

This structure contains the state of a single instance of a SoftUART module.

## 17.2.2 Function Documentation

### 17.2.2.1 SoftUARTBreakCtl

Causes a BREAK to be sent.

**Prototype:**

```
void  
SoftUARTBreakCtl(tSoftUART *psUART,  
                 bool bBreakState)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**bBreakState** controls the output level.

**Description:**

Calling this function with *bBreakState* set to **true** asserts a break condition on the SoftUART. Calling this function with *bBreakState* set to **false** removes the break condition. For proper transmission of a break command, the break must be asserted for at least two complete frames.

**Returns:**

None.

### 17.2.2.2 SoftUARTBusy

Determines whether the UART transmitter is busy or not.

**Prototype:**

```
bool  
SoftUARTBusy(tSoftUART *psUART)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**Description:**

Allows the caller to determine whether all transmitted bytes have cleared the transmitter hardware. If **false** is returned, the transmit buffer 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.

### 17.2.2.3 SoftUARTCallbackSet

Sets the callback used by the SoftUART module.

**Prototype:**

```
void  
SoftUARTCallbackSet (tSoftUART *psUART,  
                     void (*pfnCallback) (void))
```

**Parameters:**

**psUART** specifies the SoftUART data structure.  
**pfnCallback** is a pointer to the callback function.

**Description:**

This function sets the address of the callback function that is called when there is an “interrupt” produced by the SoftUART module.

**Returns:**

None.

### 17.2.2.4 SoftUARTCharGet

Waits for a character from the specified port.

**Prototype:**

```
int32_t  
SoftUARTCharGet (tSoftUART *psUART)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**Description:**

Gets a character from the receive buffer for the specified port. If there are no characters available, this function waits until a character is received before returning.

**Returns:**

Returns the character read from the specified port, cast as a *int32\_t*.

### 17.2.2.5 SoftUARTCharGetNonBlocking

Receives a character from the specified port.

**Prototype:**

```
int32_t  
SoftUARTCharGetNonBlocking (tSoftUART *psUART)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**Description:**

Gets a character from the receive buffer for the specified port.

**Returns:**

Returns the character read from the specified port, cast as a *int32\_t*. A **-1** is returned if there are no characters present in the receive buffer. The [SoftUARTCharsAvail\(\)](#) function should be called before attempting to call this function.

### 17.2.2.6 SoftUARTCharPut

Waits to send a character from the specified port.

**Prototype:**

```
void  
SoftUARTCharPut (tSoftUART *psUART,  
                 uint8_t ui8Data)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**ui8Data** is the character to be transmitted.

**Description:**

Sends the character *ui8Data* to the transmit buffer for the specified port. If there is no space available in the transmit buffer, this function waits until there is space available before returning.

**Returns:**

None.

### 17.2.2.7 SoftUARTCharPutNonBlocking

Sends a character to the specified port.

**Prototype:**

```
bool  
SoftUARTCharPutNonBlocking (tSoftUART *psUART,  
                             uint8_t ui8Data)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**ui8Data** is the character to be transmitted.

**Description:**

Writes the character *ui8Data* to the transmit buffer for the specified port. This function does not block, so if there is no space available, then a **false** is returned, and the application must retry the function later.

**Returns:**

Returns **true** if the character was successfully placed in the transmit buffer or **false** if there was no space available in the transmit buffer.

### 17.2.2.8 SoftUARTCharsAvail

Determines if there are any characters in the receive buffer.

**Prototype:**

```
bool  
SoftUARTCharsAvail (tSoftUART *psUART)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**Description:**

This function returns a flag indicating whether or not there is data available in the receive buffer.

**Returns:**

Returns **true** if there is data in the receive buffer or **false** if there is no data in the receive buffer.

### 17.2.2.9 SoftUARTConfigGet

Gets the current configuration of a UART.

**Prototype:**

```
void  
SoftUARTConfigGet (tSoftUART *psUART,  
                   uint32_t *pui32Config)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**pui32Config** is a pointer to storage for the data format.

**Description:**

Returns the data format of the SoftUART. The data format returned in *pui32Config* is enumerated the same as the *ui32Config* parameter of [SoftUARTConfigSet\(\)](#).

**Returns:**

None.

### 17.2.2.10 SoftUARTConfigSet

Sets the configuration of a SoftUART module.

**Prototype:**

```
void  
SoftUARTConfigSet (tSoftUART *psUART,  
                   uint32_t ui32Config)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**ui32Config** is the data format for the port (number of data bits, number of stop bits, and parity).

**Description:**

This function configures the SoftUART for operation in the specified data format, as specified in the *ui32Config* parameter.

The *ui32Config* parameter is the logical OR of three values: the number of data bits, the number of stop bits, and the parity. **SOFTUART\_CONFIG\_WLEN\_8**, **SOFTUART\_CONFIG\_WLEN\_7**, **SOFTUART\_CONFIG\_WLEN\_6**, and **SOFTUART\_CONFIG\_WLEN\_5** select from eight to five data bits per byte (respectively). **SOFTUART\_CONFIG\_STOP\_ONE** and **SOFTUART\_CONFIG\_STOP\_TWO** select one or two stop bits (respectively). **SOFTUART\_CONFIG\_PAR\_NONE**, **SOFTUART\_CONFIG\_PAR\_EVEN**, **SOFTUART\_CONFIG\_PAR\_ODD**, **SOFTUART\_CONFIG\_PAR\_ONE**, and **SOFTUART\_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).

**Returns:**

None.

### 17.2.2.11 SoftUARTDisable

Disables the SoftUART.

**Prototype:**

```
void  
SoftUARTDisable(tSoftUART *psUART)
```

**Parameters:**

***psUART*** specifies the SoftUART data structure.

**Description:**

This function disables the SoftUART after waiting for it to become idle.

**Returns:**

None.

### 17.2.2.12 SoftUARTEnable

Enables the SoftUART.

**Prototype:**

```
void  
SoftUARTEnable(tSoftUART *psUART)
```

**Parameters:**

***psUART*** specifies the SoftUART data structure.

**Description:**

This function enables the SoftUART, allowing data to be transmitted and received.

**Returns:**

None.

### 17.2.2.13 SoftUARTFIFOLevelGet

Gets the buffer level at which “interrupts” are generated.

**Prototype:**

```
void  
SoftUARTFIFOLevelGet (tSoftUART *psUART,  
                      uint32_t *pui32TxLevel,  
                      uint32_t *pui32RxLevel)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**pui32TxLevel** is a pointer to storage for the transmit buffer 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**.

**pui32RxLevel** is a pointer to storage for the receive buffer 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 buffer level at which transmit and receive “interrupts” are generated.

**Returns:**

None.

### 17.2.2.14 SoftUARTFIFOLevelSet

Sets the buffer level at which “interrupts” are generated.

**Prototype:**

```
void  
SoftUARTFIFOLevelSet (tSoftUART *psUART,  
                      uint32_t ui32TxLevel,  
                      uint32_t ui32RxLevel)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**ui32TxLevel** is the transmit buffer “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**.

**ui32RxLevel** is the receive buffer “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 buffer level at which transmit and receive “interrupts” are generated.

**Returns:**

None.



### 17.2.2.15 SoftUARTInit

Initializes the SoftUART module.

**Prototype:**

```
void  
SoftUARTInit (tSoftUART *psUART)
```

**Parameters:**

**psUART** specifies the soft UART data structure.

**Description:**

This function initializes the data structure for the SoftUART module, putting it into the default configuration.

**Returns:**

None.

### 17.2.2.16 SoftUARTIntClear

Clears SoftUART “interrupt” sources.

**Prototype:**

```
void  
SoftUARTIntClear (tSoftUART *psUART,  
                  uint32_t ui32IntFlags)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**ui32IntFlags** is a bit mask of the “interrupt” sources to be cleared.

**Description:**

The specified SoftUART “interrupt” sources are cleared, so that they no longer assert. This function must be called in the callback function to keep the “interrupt” from being recognized again immediately upon exit.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [SoftUARTIntEnable\(\)](#).

**Returns:**

None.

### 17.2.2.17 SoftUARTIntDisable

Disables individual SoftUART “interrupt” sources.

**Prototype:**

```
void  
SoftUARTIntDisable (tSoftUART *psUART,  
                    uint32_t ui32IntFlags)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**ui32IntFlags** is the bit mask of the “interrupt” sources to be disabled.

**Description:**

Disables the indicated SoftUART “interrupt” sources. Only the sources that are enabled can be reflected to the SoftUART callback.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to [SoftUARTIntEnable\(\)](#).

**Returns:**

None.

### 17.2.2.18 SoftUARTIntEnable

Enables individual SoftUART “interrupt” sources.

**Prototype:**

```
void  
SoftUARTIntEnable(tSoftUART *psUART,  
                  uint32_t ui32IntFlags)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**ui32IntFlags** is the bit mask of the “interrupt” sources to be enabled.

**Description:**

Enables the indicated SoftUART “interrupt” sources. Only the sources that are enabled can be reflected to the SoftUART callback.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **SOFTUART\_INT\_OE** - Overrun Error “interrupt”
- **SOFTUART\_INT\_BE** - Break Error “interrupt”
- **SOFTUART\_INT\_PE** - Parity Error “interrupt”
- **SOFTUART\_INT\_FE** - Framing Error “interrupt”
- **SOFTUART\_INT\_RT** - Receive Timeout “interrupt”
- **SOFTUART\_INT\_TX** - Transmit “interrupt”
- **SOFTUART\_INT\_RX** - Receive “interrupt”

**Returns:**

None.

### 17.2.2.19 SoftUARTIntStatus

Gets the current SoftUART “interrupt” status.

**Prototype:**

```
uint32_t  
SoftUARTIntStatus(tSoftUART *psUART,  
                  bool bMasked)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**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 SoftUART. Either the raw “interrupt” status or the status of “interrupts” that are allowed to reflect to the SoftUART callback can be returned.

**Returns:**

Returns the current “interrupt” status, enumerated as a bit field of values described in [SoftUARTIntEnable\(\)](#).

### 17.2.2.20 SoftUARTParityModeGet

Gets the type of parity currently being used.

**Prototype:**

```
uint32_t  
SoftUARTParityModeGet (tSoftUART *psUART)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**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 **SOFTUART\_CONFIG\_PAR\_NONE**, **SOFTUART\_CONFIG\_PAR\_EVEN**, **SOFTUART\_CONFIG\_PAR\_ODD**, **SOFTUART\_CONFIG\_PAR\_ONE**, or **SOFTUART\_CONFIG\_PAR\_ZERO**.

### 17.2.2.21 SoftUARTParityModeSet

Sets the type of parity.

**Prototype:**

```
void  
SoftUARTParityModeSet (tSoftUART *psUART,  
                        uint32_t ui32Parity)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**ui32Parity** specifies the type of parity to use.

**Description:**

Sets the type of parity to use for transmitting and expect when receiving. The *ui32Parity* parameter must be one of **SOFTUART\_CONFIG\_PAR\_NONE**, **SOFTUART\_CONFIG\_PAR\_EVEN**, **SOFTUART\_CONFIG\_PAR\_ODD**, **SOFTUART\_CONFIG\_PAR\_ONE**, or **SOFTUART\_CONFIG\_PAR\_ZERO**. The last two allow direct control of the parity bit; it is always either one or zero based on the mode.

**Returns:**

None.

#### 17.2.2.22 SoftUARTRxBufferSet

Sets the receive buffer for a SoftUART module.

**Prototype:**

```
void  
SoftUARTRxBufferSet (tSoftUART *psUART,  
                     uint16_t *pui16RxBuffer,  
                     uint16_t ui16Len)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**pui16RxBuffer** is the address of the receive buffer.

**ui16Len** is the size, in 16-bit half-words, of the receive buffer.

**Description:**

This function sets the address and size of the receive buffer. It also resets the read and write pointers, marking the receive buffer as empty.

**Returns:**

None.

#### 17.2.2.23 SoftUARTRxErrorClear

Clears all reported receiver errors.

**Prototype:**

```
void  
SoftUARTRxErrorClear (tSoftUART *psUART)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**Description:**

This function is used to clear all receiver error conditions reported via [SoftUARTRxErrorGet\(\)](#). 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.

#### 17.2.2.24 SoftUARTRxErrorGet

Gets current receiver errors.

**Prototype:**

```
uint32_t
SoftUARTRxErrorGet (tSoftUART *psUART)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**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 [SoftUARTCharGet\(\)](#) or [SoftUARTCharGetNonBlocking\(\)](#) with the exception that the overrun error is set immediately when the overrun occurs rather than when a character is next read.

**Returns:**

Returns a logical OR combination of the receiver error flags, **SOFTUART\_RXERROR\_FRAMING**, **SOFTUART\_RXERROR\_PARITY**, **SOFTUART\_RXERROR\_BREAK** and **SOFTUART\_RXERROR\_OVERRUN**.

### 17.2.2.25 SoftUARTRxGPIOSet

Sets the GPIO pin to be used as the SoftUART Rx signal.

**Prototype:**

```
void
SoftUARTRxGPIOSet (tSoftUART *psUART,
                   uint32_t ui32Base,
                   uint8_t ui8Pin)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**ui32Base** is the base address of the GPIO module.

**ui8Pin** is the bit-packed representation of the pin to use.

**Description:**

This function sets the GPIO pin that is used when the SoftUART must sample the Rx signal. If there is not a GPIO pin allocated for Rx, the SoftUART module will not read data from the slave device.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None.

### 17.2.2.26 SoftUARTRxTick

Performs the periodic update of the SoftUART receiver.

**Prototype:**

```
uint32_t
SoftUARTRxTick (tSoftUART *psUART,
                bool bEdgeInt)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**bEdgeInt** should be **true** if this function is being called because of a GPIO edge interrupt and **false** if it is being called because of a timer interrupt.

**Description:**

This function performs the periodic, time-based updates to the SoftUART receiver. The reception of data to the SoftUART is performed by the state machine in this function.

This function must be called by the GPIO interrupt handler, and then periodically at the desired SoftUART baud rate. For example, to run the SoftUART at 115, 200 baud, this function must be called at a 115, 200 Hz rate.

**Returns:**

Returns **SOFTUART\_RXTIMER\_NOP** if the receive timer should continue to operate or **SOFTUART\_RXTIMER\_END** if it should be stopped.

### 17.2.2.27 SoftUARTSpaceAvail

Determines if there is any space in the transmit buffer.

**Prototype:**

```
bool  
SoftUARTSpaceAvail (tSoftUART *psUART)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**Description:**

This function returns a flag indicating whether or not there is space available in the transmit buffer.

**Returns:**

Returns **true** if there is space available in the transmit buffer or **false** if there is no space available in the transmit buffer.

### 17.2.2.28 SoftUARTTxBufferSet

Sets the transmit buffer for a SoftUART module.

**Prototype:**

```
void  
SoftUARTTxBufferSet (tSoftUART *psUART,  
                     uint8_t *pui8TxBuffer,  
                     uint16_t ui16Len)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**pui8TxBuffer** is the address of the transmit buffer.

**ui16Len** is the size, in 8-bit bytes, of the transmit buffer.

**Description:**

This function sets the address and size of the transmit buffer. It also resets the read and write pointers, marking the transmit buffer as empty.

**Returns:**

None.

### 17.2.2.29 SoftUARTTxGPIOSet

Sets the GPIO pin to be used as the SoftUART Tx signal.

**Prototype:**

```
void  
SoftUARTTxGPIOSet (tSoftUART *psUART,  
                   uint32_t ui32Base,  
                   uint8_t ui8Pin)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**ui32Base** is the base address of the GPIO module.

**ui8Pin** is the bit-packed representation of the pin to use.

**Description:**

This function sets the GPIO pin that is used when the SoftUART must assert the Tx signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

**Returns:**

None.

### 17.2.2.30 SoftUARTTxTimerTick

Performs the periodic update of the SoftUART transmitter.

**Prototype:**

```
void  
SoftUARTTxTimerTick (tSoftUART *psUART)
```

**Parameters:**

**psUART** specifies the SoftUART data structure.

**Description:**

This function performs the periodic, time-based updates to the SoftUART transmitter. The transmission of data from the SoftUART is performed by the state machine in this function.

This function must be called at the desired SoftUART baud rate. For example, to run the SoftUART at 115, 200 baud, this function must be called at a 115, 200 Hz rate.

**Returns:**

None.

## 17.3 Programming Example

The following example shows how to configure the software UART module and transmit some data to an external peripheral. This example uses Timer 0 as the timing source.

```
//
// The instance data for the software UART.
//
tSoftUART g_sUART;

//
// The buffer used to hold the transmit data.
//
unsigned char g_pucTxBuffer[16];

//
// The buffer used to hold the receive data.
//
unsigned short g_pusRxBuffer[16];

//
// The number of processor clocks in the time period of a single bit on the
// software UART interface.
//
unsigned long g_ulBitTime;

//
// The transmit timer tick function.
//
void
Timer0AIntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Call the software UART transmit timer tick function.
    //
    SoftUARTTxTimerTick(&g_sUART);
}

//
// The receive timer tick function.
//
void
Timer0BIntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Call the software UART receive timer tick function, and see if the
    // timer should be disabled.
    //
    if(SoftUARTRxTick(&g_sUART, false) == SOFTUART_RXTIMER_END)
    {
        //
        // Disable the timer interrupt since the software UART doesn't need
        // it any longer.
        //
    }
}
```



```
        TimerDisable(TIMER0_BASE, TIMER_B);
    }
}

//
// The interrupt handler for the software UART GPIO edge interrupt.
//
void
GPIOIntHandler(void)
{
    //
    // Configure the software UART receive timer so that it samples at the
    // mid-bit time of this character.
    //
    TimerDisable(TIMER0_BASE, TIMER_B);
    TimerLoadSet(TIMER0_BASE, TIMER_B, g_ulBitTime);
    TimerIntClear(TIMER0_BASE, TIMER_TIMB_TIMEOUT);
    TimerEnable(TIMER0_BASE, TIMER_B);

    //
    // Call the software UART receive timer tick function.
    //
    SoftUARTRxTick(&g_sUART, true);
}

//
// The callback function for the software UART. This function is
// equivalent to the interrupt handler for a hardware UART.
//
void
UARTCallback(void)
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = SoftUARTIntStatus(&g_sUART, true);

    //
    // Clear the asserted interrupt sources.
    //
    SoftUARTIntClear(&g_sUART, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the software UART and send some data.
//
void
TestSoftUART(void)
{
    //
    // Initialize the software UART instance data.
    //
    SoftUARTInit(&g_sUART);

    //
    // Set the callback function used for this software UART.
    //
    SoftUARTCallbackSet(&g_sUART, UARTCallback);
}
```

```
//
// Configure the pins used for the software UART. This example uses
// pins PD0 and PE1.
//
SoftUARTTxGPIOSet(&g_sUART, GPIO_PORTD_BASE, GPIO_PIN_0);
SoftUARTRxGPIOSet(&g_sUART, GPIO_PORTE_BASE, GPIO_PIN_1);

//
// Configure the data buffers used as the transmit and receive buffers.
//
SoftUARTTxBufferSet(&g_sUART, g_pucTxBuffer, 16);
SoftUARTRxBufferSet(&g_sUART, g_pusRxBuffer, 16);

//
// Enable the GPIO modules that contains the GPIO pins to be used by
// the software UART.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

//
// Configure the software UART module: 8 data bits, no parity, and one
// stop bit.
//
SoftUARTConfigSet(&g_sUART,
                  (SOFTUART_CONFIG_WLEN_8 | SOFTUART_CONFIG_PAR_NONE |
                   SOFTUART_CONFIG_STOP_ONE));

//
// Compute the bit time for 38,400 baud.
//
g_ulBitTime = (SysCtlClockGet() / 38400) - 1;

//
// Configure the timers used to generate the timing for the software
// UART. The interface in this example is run at 38,400 baud,
// requiring a timer tick at 38,400 Hz.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerConfigure(TIMER0_BASE,
               (TIMER_CFG_16_BIT_PAIR | TIMER_CFG_A_PERIODIC |
                TIMER_CFG_B_PERIODIC));
TimerLoadSet(TIMER0_BASE, TIMER_A, g_ulBitTime);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT | TIMER_TIMB_TIMEOUT);
TimerEnable(TIMER0_BASE, TIMER_A);

//
// Set the priorities of the interrupts associated with the software
// UART. The receiver is higher priority than the transmitter, and the
// receiver edge interrupt is higher priority than the receiver timer
// interrupt.
//
IntPrioritySet(INT_GPIOE, 0x00);
IntPrioritySet(INT_TIMER0B, 0x40);
IntPrioritySet(INT_TIMER0A, 0x80);

//
// Enable the interrupts associated with the software UART.
//
IntEnable(INT_GPIOE);
IntEnable(INT_TIMER0A);
IntEnable(INT_TIMER0B);

//
// Enable the transmit FIFO half full interrupt in the software UART.
//
```

```
SoftUARTIntEnable(&g_sUART, SOFTUART_INT_TX);

//
// Write some data into the software UART transmit FIFO.
//
SoftUARTCharPut(&g_sUART, 0x55);
SoftUARTCharPut(&g_sUART, 0xaa);
SoftUARTCharPut(&g_sUART, 0x55);
SoftUARTCharPut(&g_sUART, 0xaa);
SoftUARTCharPut(&g_sUART, 0x55);
SoftUARTCharPut(&g_sUART, 0xaa);
SoftUARTCharPut(&g_sUART, 0x55);
SoftUARTCharPut(&g_sUART, 0xaa);
SoftUARTCharPut(&g_sUART, 0x55);
SoftUARTCharPut(&g_sUART, 0xaa);
SoftUARTCharPut(&g_sUART, 0x55);
SoftUARTCharPut(&g_sUART, 0xaa);

//
// Wait until the software UART is idle. The transmit FIFO half full
// interrupt is sent to the callback function prior to exiting this
// loop.
//
while(SoftUARTBusy(&g_sUART))
{
}
}
```

As a comparison, the following is the equivalent code using the hardware UART module and the Tiva Peripheral Driver Library.

```
//
// The interrupt handler for the hardware UART.
//
void
UART0IntHandler(void)
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = UARTIntStatus(UART0_BASE, true);

    //
    // Clear the asserted interrupt sources.
    //
    UARTIntClear(UART0_BASE, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the hardware UART and send some data.
//
void
TestUART(void)
{
    //
    // Enable the GPIO module that contains the GPIO pins to be used by
    // the UART, as well as the UART module.
    //
}
```

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

//
// Configure the GPIO pins for use by the UART module.
//
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

//
// Initialize the hardware UART module: 8 data bits, no parity, one stop
// bit, and 38,400 baud rate.
//
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 38400,
                    (UART_CONFIG_WLEN_8 | UART_CONFIG_PAR_NONE |
                     UART_CONFIG_STOP_ONE));

//
// Enable the transmit FIFO half full interrupt in the hardware UART.
//
UARTIntEnable(UART0_BASE, UART_INT_TX);
IntEnable(INT_UART0);

//
// Write some data into the hardware UART transmit FIFO.
//
UARTCharPut(UART0_BASE, 0x55);
UARTCharPut(UART0_BASE, 0xaa);
UARTCharPut(UART0_BASE, 0x55);
UARTCharPut(UART0_BASE, 0xaa);
UARTCharPut(UART0_BASE, 0x55);
UARTCharPut(UART0_BASE, 0xaa);
UARTCharPut(UART0_BASE, 0x55);
UARTCharPut(UART0_BASE, 0xaa);
UARTCharPut(UART0_BASE, 0x55);
UARTCharPut(UART0_BASE, 0xaa);
UARTCharPut(UART0_BASE, 0x55);
UARTCharPut(UART0_BASE, 0xaa);

//
// Wait until the hardware UART is idle. The transmit FIFO half full
// interrupt is sent to the interrupt handler prior to exiting this
// loop.
//
while(UARTBusy(UART0_BASE))
{
}
}
```

## 18 SPI Flash Module

Introduction .....	149
API Functions .....	149
Programming Example .....	163

### 18.1 Introduction

This module provides for configuring, reading and programming an external memory device that is connected to a SSI port (SPI flash).

Prior to using the SPI Flash API, the application must enable the SSI peripheral and configure the appropriate GPIO pins for use by the SSI. Once that has been done, then the SSI peripheral is configured for use with external flash by calling [SPIFlashInit\(\)](#). The status of the external memory can be checked by calling [SPIFlashReadStatus\(\)](#) and the ID of the device can be read with [SPIFlashReadID\(\)](#).

There are a set of functions for reading from flash. [SPIFlashRead\(\)](#), [SPIFlashFastRead\(\)](#), [SPIFlashDualRead\(\)](#) and [SPIFlashQuadRead\(\)](#) are used for reading flash using normal, fast, Bi-SPI and Quad-SPI modes, respectively.

The external flash can be erased using [SPIFlashSectorErase\(\)](#), [SPIFlashChipErase\(\)](#), [SPIFlashBlockErase32\(\)](#) and [SPIFlashBlockErase64\(\)](#).

The flash can be programmed using [SPIFlashPageProgram\(\)](#).

All of the reading and programming functions mentioned so far are "blocking". This means that when the function is called it will not return until the operation is complete, and there will be polling loops during which no other processing can take place. For these functions, they each also have a non-blocking form which has the same name with "NonBlocking" appended. For example [SPIFlashReadNonBlocking\(\)](#) is used to perform reads in a non-blocking manner.

To perform non-blocking operation, the uDMA controller is used to perform transfers in the background, with SSI interrupt occurring after each block has been transferred. The application must implement an interrupt handler for the SSI peripheral, and whenever it is triggered, must call the SPI flash handler named [SPIFlashIntHandler\(\)](#). This function will process an ongoing non-blocking transfer and will then return to the caller with an indication that the transfer is still ongoing or is complete. While a non-blocking transfer is taking place, other non-SSI related code can be executed, allowing for a parallel operation during a block read or write.

This module is contained in `utils/spi_flash.c`, with `utils/spi_flash.h` containing the API declarations for use by applications.

### 18.2 API Functions

#### Data Structures

- [tSPIFlashState](#)

## Functions

- void [SPIFlashBlockErase32](#) (uint32\_t ui32Base, uint32\_t ui32Addr)
- void [SPIFlashBlockErase64](#) (uint32\_t ui32Base, uint32\_t ui32Addr)
- void [SPIFlashChipErase](#) (uint32\_t ui32Base)
- void [SPIFlashDualRead](#) (uint32\_t ui32Base, uint32\_t ui32Addr, uint8\_t \*pui8Data, uint32\_t ui32Count)
- void [SPIFlashDualReadNonBlocking](#) (tSPIFlashState \*pState, uint32\_t ui32Base, uint32\_t ui32Addr, uint8\_t \*pui8Data, uint32\_t ui32Count, bool bUseDMA, uint32\_t ui32TxChannel, uint32\_t ui32RxChannel)
- void [SPIFlashFastRead](#) (uint32\_t ui32Base, uint32\_t ui32Addr, uint8\_t \*pui8Data, uint32\_t ui32Count)
- void [SPIFlashFastReadNonBlocking](#) (tSPIFlashState \*pState, uint32\_t ui32Base, uint32\_t ui32Addr, uint8\_t \*pui8Data, uint32\_t ui32Count, bool bUseDMA, uint32\_t ui32TxChannel, uint32\_t ui32RxChannel)
- void [SPIFlashInit](#) (uint32\_t ui32Base, uint32\_t ui32Clock, uint32\_t ui32BitRate)
- uint32\_t [SPIFlashIntHandler](#) (tSPIFlashState \*pState)
- void [SPIFlashPageProgram](#) (uint32\_t ui32Base, uint32\_t ui32Addr, const uint8\_t \*pui8Data, uint32\_t ui32Count)
- void [SPIFlashPageProgramNonBlocking](#) (tSPIFlashState \*pState, uint32\_t ui32Base, uint32\_t ui32Addr, const uint8\_t \*pui8Data, uint32\_t ui32Count, bool bUseDMA, uint32\_t ui32TxChannel)
- void [SPIFlashQuadRead](#) (uint32\_t ui32Base, uint32\_t ui32Addr, uint8\_t \*pui8Data, uint32\_t ui32Count)
- void [SPIFlashQuadReadNonBlocking](#) (tSPIFlashState \*pState, uint32\_t ui32Base, uint32\_t ui32Addr, uint8\_t \*pui8Data, uint32\_t ui32Count, bool bUseDMA, uint32\_t ui32TxChannel, uint32\_t ui32RxChannel)
- void [SPIFlashRead](#) (uint32\_t ui32Base, uint32\_t ui32Addr, uint8\_t \*pui8Data, uint32\_t ui32Count)
- void [SPIFlashReadID](#) (uint32\_t ui32Base, uint8\_t \*pui8ManufacturerID, uint16\_t \*pui16DeviceID)
- void [SPIFlashReadNonBlocking](#) (tSPIFlashState \*pState, uint32\_t ui32Base, uint32\_t ui32Addr, uint8\_t \*pui8Data, uint32\_t ui32Count, bool bUseDMA, uint32\_t ui32TxChannel, uint32\_t ui32RxChannel)
- uint8\_t [SPIFlashReadStatus](#) (uint32\_t ui32Base)
- void [SPIFlashSectorErase](#) (uint32\_t ui32Base, uint32\_t ui32Addr)
- void [SPIFlashWriteDisable](#) (uint32\_t ui32Base)
- void [SPIFlashWriteEnable](#) (uint32\_t ui32Base)
- void [SPIFlashWriteStatus](#) (uint32\_t ui32Base, uint8\_t ui8Status)

## 18.2.1 Data Structure Documentation

### 18.2.1.1 tSPIFlashState

**Definition:**

```
typedef struct
{
    uint32_t ui32Base;
```

```

uint16_t ui16Cmd;
uint16_t ui16State;
uint32_t ui32Addr;
uint8_t *pui8Buffer;
uint32_t ui32ReadCount;
uint32_t ui32WriteCount;
bool bUseDMA;
uint32_t ui32TxChannel;
uint32_t ui32RxChannel;
}
tSPIFlashState

```

**Members:**

**ui32Base** The base address of the SSI module that is being used.

**ui16Cmd** The command that is being send to the SPI flash.

**ui16State** The current state of the SPI flash state machine.

**ui32Addr** The SPI flash address associated with the command.

**pui8Buffer** A pointer to the data buffer that is being read or written.

**ui32ReadCount** The count of bytes left to be read.

**ui32WriteCount** The count of bytes left to be written.

**bUseDMA** A flag that is true if uDMA used be used for the transfer.

**ui32TxChannel** The uDMA channel to use for transmitting when using uDMA for the transfer.

**ui32RxChannel** The uDMA channel to use for receiving when using uDMA for the transfer.

**Description:**

The state structure used when performing non-blocking SPI flash operations.

## 18.2.2 Function Documentation

### 18.2.2.1 SPIFlashBlockErase32

Erases a 32 KB block of the SPI flash.

**Prototype:**

```

void
SPIFlashBlockErase32(uint32_t ui32Base,
                     uint32_t ui32Addr)

```

**Parameters:**

**ui32Base** is the SSI module base address.

**ui32Addr** is the SPI flash address to erase.

**Description:**

This function erases a 32 KB block of the SPI flash. Each 32 KB block has a 32 KB alignment; the SPI flash will ignore the lower 15 bits of the address provided. The 32 KB block erase command is issued by this function; [SPIFlashReadStatus\(\)](#) must be used to query the SPI flash to determine when the 32 KB block erase operation has completed. This uses the 0x52 SPI flash command.

**Returns:**

None.

### 18.2.2.2 SPIFlashBlockErase64

Erases a 64 KB block of the SPI flash.

**Prototype:**

```
void  
SPIFlashBlockErase64(uint32_t ui32Base,  
                     uint32_t ui32Addr)
```

**Parameters:**

***ui32Base*** is the SSI module base address.

***ui32Addr*** is the SPI flash address to erase.

**Description:**

This function erases a 64 KB block of the SPI flash. Each 64 KB block has a 64 KB alignment; the SPI flash will ignore the lower 16 bits of the address provided. The 64 KB block erase command is issued by this function; [SPIFlashReadStatus\(\)](#) must be used to query the SPI flash to determine when the 64 KB block erase operation has completed. This uses the 0xd8 SPI flash command.

**Returns:**

None.

### 18.2.2.3 SPIFlashChipErase

Erases the entire SPI flash.

**Prototype:**

```
void  
SPIFlashChipErase(uint32_t ui32Base)
```

**Parameters:**

***ui32Base*** is the SSI module base address.

**Description:**

This command erase the entire SPI flash. The chip erase command is issued by this function; [SPIFlashReadStatus\(\)](#) must be used to query the SPI flash to determine when the chip erase operation has completed. This uses the 0xc7 SPI flash command.

**Returns:**

None.

### 18.2.2.4 SPIFlashDualRead

Reads data from the SPI flash using Bi-SPI.

**Prototype:**

```
void  
SPIFlashDualRead(uint32_t ui32Base,  
                 uint32_t ui32Addr,  
                 uint8_t *pui8Data,  
                 uint32_t ui32Count)
```



**Parameters:**

**ui32Base** is the SSI module base address.  
**ui32Addr** is the SPI flash address to read.  
**pui8Data** is a pointer to the data buffer to into which to read the data.  
**ui32Count** is the number of bytes to read.

**Description:**

This function reads data from the SPI flash with Bi-SPI, using PIO mode. This function will not return until the read has completed. This uses the 0x3b SPI flash command.

**Returns:**

None.

### 18.2.2.5 SPIFlashDualReadNonBlocking

Reads data from the SPI flash using Bi-SPI in the background.

**Prototype:**

```
void
SPIFlashDualReadNonBlocking(tSPIFlashState *pState,
                             uint32_t ui32Base,
                             uint32_t ui32Addr,
                             uint8_t *pui8Data,
                             uint32_t ui32Count,
                             bool bUseDMA,
                             uint32_t ui32TxChannel,
                             uint32_t ui32RxChannel)
```

**Parameters:**

**pState** is a pointer to the SPI flash state structure.  
**ui32Base** is the SSI module base address.  
**ui32Addr** is the SPI flash address to read.  
**pui8Data** is a pointer to the data buffer to into which to read the data.  
**ui32Count** is the number of bytes to read.  
**bUseDMA** is **true** if uDMA should be used and **false** otherwise.  
**ui32TxChannel** is the uDMA channel to be used for writing to the SSI module.  
**ui32RxChannel** is the uDMA channel to be used for reading from the SSI module.

**Description:**

This function reads data from the SPI flash with Bi-SPI, using either interrupts or uDMA to transfer the data. This function will return immediately and read the data in the background. In order for this to complete successfully, several conditions must be satisfied:

- Prior to calling this function:
  - The SSI module must be enabled in SysCtl.
  - The SSI pins must be configured for use by the SSI module.
  - The SSI module interrupt must be enabled in NVIC.
  - The uDMA module must be enabled in SysCtl and the control table set (if using uDMA).
  - The uDMA channels must be assigned to the SSI module.

■ After calling this function:

- The interrupt handler for the SSI module must call [SPIFlashIntHandler\(\)](#), passing the same pState structure pointer that was supplied to this function.
- No other SPI flash operation can be called until this operation has completed.

Completion of the read operation is indicated when [SPIFlashIntHandler\(\)](#) returns **SPI\_FLASH\_DONE**.

Like SPIFlashDualRead(), this uses the 0x3b SPI flash command.

**Returns:**

None.

### 18.2.2.6 SPIFlashFastRead

Reads data from the SPI flash using the fast read command.

**Prototype:**

```
void
SPIFlashFastRead(uint32_t ui32Base,
                 uint32_t ui32Addr,
                 uint8_t *pui8Data,
                 uint32_t ui32Count)
```

**Parameters:**

**ui32Base** is the SSI module base address.

**ui32Addr** is the SPI flash address to read.

**pui8Data** is a pointer to the data buffer to into which to read the data.

**ui32Count** is the number of bytes to read.

**Description:**

This function reads data from the SPI flash with the fast read command, using PIO mode. The fast read command allows the SPI flash to be read at a higher SPI clock rate because of the addition of a dummy cycle during the command setup. This function will not return until the read has completed. This uses the 0x0b SPI flash command.

**Returns:**

None.

### 18.2.2.7 SPIFlashFastReadNonBlocking

Reads data from the SPI flash using the fast read command in the background.

**Prototype:**

```
void
SPIFlashFastReadNonBlocking(tSPIFlashState *pState,
                           uint32_t ui32Base,
                           uint32_t ui32Addr,
                           uint8_t *pui8Data,
                           uint32_t ui32Count,
```

```
bool bUseDMA,
uint32_t ui32TxChannel,
uint32_t ui32RxChannel)
```

**Parameters:**

**pState** is a pointer to the SPI flash state structure.  
**ui32Base** is the SSI module base address.  
**ui32Addr** is the SPI flash address to read.  
**pui8Data** is a pointer to the data buffer to into which to read the data.  
**ui32Count** is the number of bytes to read.  
**bUseDMA** is **true** if uDMA should be used and **false** otherwise.  
**ui32TxChannel** is the uDMA channel to be used for writing to the SSI module.  
**ui32RxChannel** is the uDMA channel to be used for reading from the SSI module.

**Description:**

This function reads data from the SPI flash with the fast read command, using either interrupts or uDMA to transfer the data. The fast read command allows the SPI flash to be read at a higher SPI clock rate because of the addition of a dummy cycle during the command setup. This function will return immediately and read the data in the background. In order for this to complete successfully, several conditions must be satisfied:

- Prior to calling this function:
  - The SSI module must be enabled in SysCtl.
  - The SSI pins must be configured for use by the SSI module.
  - The SSI module interrupt must be enabled in NVIC.
  - The uDMA module must be enabled in SysCtl and the control table set (if using uDMA).
  - The uDMA channels must be assigned to the SSI module.
- After calling this function:
  - The interrupt handler for the SSI module must call [SPIFlashIntHandler\(\)](#), passing the same pState structure pointer that was supplied to this function.
  - No other SPI flash operation can be called until this operation has completed.

Completion of the read operation is indicated when [SPIFlashIntHandler\(\)](#) returns **SPI\_FLASH\_DONE**.

Like [SPIFlashFastRead\(\)](#), this uses the 0x0b SPI flash command.

**Returns:**

None.

### 18.2.2.8 SPIFlashInit

Initializes the SPI flash driver.

**Prototype:**

```
void
SPIFlashInit (uint32_t ui32Base,
              uint32_t ui32Clock,
              uint32_t ui32BitRate)
```

**Parameters:**

**ui32Base** is the SSI module base address.

**ui32Clock** is the rate of the clock supplied to the SSI module.

**ui32BitRate** is the SPI clock rate.

**Description:**

This function configures the SSI module for use by the SPI flash driver. The SSI module will be placed into the correct mode of operation to allow communication with the SPI flash. This function must be called prior to calling the remaining SPI flash driver APIs. It can be called at a later point to reconfigure the SSI module, such as to increase the SPI clock rate once it has been determined that it is safe to use a higher speed clock.

It is the responsibility of the caller to enable the SSI module and configure the pins that it will utilize.

**Returns:**

None.

### 18.2.2.9 SPIFlashIntHandler

Handles SSI module interrupts for the SPI flash driver.

**Prototype:**

```
uint32_t  
SPIFlashIntHandler(tSPIFlashState *pState)
```

**Parameters:**

**pState** is a pointer to the SPI flash driver instance data.

**Description:**

This function handles SSI module interrupts that are generated as a result of SPI flash driver operations. This must be called by the application in response to the SSI module interrupt when using the SPIFlashxxxNonBlocking APIs.

**Returns:**

Returns **SPI\_FLASH\_IDLE** if there is no transfer in progress, **SPI\_FLASH\_WORKING** if the requested transfer is still in progress, or **SPI\_FLASH\_DONE** if the requested transfer has completed.

### 18.2.2.10 SPIFlashPageProgram

Programs the SPI flash.

**Prototype:**

```
void  
SPIFlashPageProgram(uint32_t ui32Base,  
                    uint32_t ui32Addr,  
                    const uint8_t *pui8Data,  
                    uint32_t ui32Count)
```

**Parameters:**

**ui32Base** is the SSI module base address.

***ui32Addr*** is the SPI flash address to be programmed.

***pui8Data*** is a pointer to the data to be programmed.

***ui32Count*** is the number of bytes to be programmed.

**Description:**

This function programs data into the SPI flash, using PIO mode. This function will not return until the entire program command has been written into the SSI transmit FIFO. This uses the 0x02 SPI flash command.

**Returns:**

None.

### 18.2.2.11 SPIFlashPageProgramNonBlocking

Programs the SPI flash in the background.

**Prototype:**

```
void
SPIFlashPageProgramNonBlocking(tSPIFlashState *pState,
                               uint32_t ui32Base,
                               uint32_t ui32Addr,
                               const uint8_t *pui8Data,
                               uint32_t ui32Count,
                               bool bUseDMA,
                               uint32_t ui32TxChannel)
```

**Parameters:**

***pState*** is a pointer to the SPI flash state structure.

***ui32Base*** is the SSI module base address.

***ui32Addr*** is the SPI flash address to be programmed.

***pui8Data*** is a pointer to the data to be programmed.

***ui32Count*** is the number of bytes to be programmed.

***bUseDMA*** is **true** if uDMA should be used and **false** otherwise.

***ui32TxChannel*** is the uDMA channel to be used for writing to the SSI module.

**Description:**

This function programs data into the SPI flash, using either interrupts or uDMA to transfer the data. This function will return immediately and send the data in the background. In order for this to complete successfully, several conditions must be satisfied:

- Prior to calling this function:
  - The SSI module must be enabled in SysCtl.
  - The SSI pins must be configured for use by the SSI module.
  - The SSI module interrupt must be enabled in NVIC.
  - The uDMA module must be enabled in SysCtl and the control table set (if using uDMA).
  - The uDMA channels must be assigned to the SSI module.
- After calling this function:
  - The interrupt handler for the SSI module must call [SPIFlashIntHandler\(\)](#), passing the same pState structure pointer that was supplied to this function.

- No other SPI flash operation can be called until this operation has completed.

Completion of the programming operation is indicated when [SPIFlashIntHandler\(\)](#) returns **SPI\_FLASH\_DONE**.

Like [SPIFlashPageProgram\(\)](#), this uses the 0x02 SPI flash command.

**Returns:**

None.

#### 18.2.2.12 SPIFlashQuadRead

Reads data from the SPI flash using Quad-SPI.

**Prototype:**

```
void
SPIFlashQuadRead(uint32_t ui32Base,
                  uint32_t ui32Addr,
                  uint8_t *pui8Data,
                  uint32_t ui32Count)
```

**Parameters:**

***ui32Base*** is the SSI module base address.

***ui32Addr*** is the SPI flash address to read.

***pui8Data*** is a pointer to the data buffer to into which to read the data.

***ui32Count*** is the number of bytes to read.

**Description:**

This function reads data from the SPI flash with Quad-SPI, using PIO mode. This function will not return until the read has completed. This uses the 0x6b SPI flash command.

**Returns:**

None.

#### 18.2.2.13 SPIFlashQuadReadNonBlocking

Reads data from the SPI flash using Quad-SPI in the background.

**Prototype:**

```
void
SPIFlashQuadReadNonBlocking(tSPIFlashState *pState,
                             uint32_t ui32Base,
                             uint32_t ui32Addr,
                             uint8_t *pui8Data,
                             uint32_t ui32Count,
                             bool bUseDMA,
                             uint32_t ui32TxChannel,
                             uint32_t ui32RxChannel)
```

**Parameters:**

***pState*** is a pointer to the SPI flash state structure.

**ui32Base** is the SSI module base address.

**ui32Addr** is the SPI flash address to read.

**pui8Data** is a pointer to the data buffer to into which to read the data.

**ui32Count** is the number of bytes to read.

**bUseDMA** is **true** if uDMA should be used and **false** otherwise.

**ui32TxChannel** is the uDMA channel to be used for writing to the SSI module.

**ui32RxChannel** is the uDMA channel to be used for reading from the SSI module.

#### Description:

This function reads data from the SPI flash with Quad-SPI, using either interrupts or uDMA to transfer the data. This function will return immediately and read the data in the background. In order for this to complete successfully, several conditions must be satisfied:

- Prior to calling this function:
  - The SSI module must be enabled in SysCtl.
  - The SSI pins must be configured for use by the SSI module.
  - The SSI module interrupt must be enabled in NVIC.
  - The uDMA module must be enabled in SysCtl and the control table set (if using uDMA).
  - The uDMA channels must be assigned to the SSI module.
- After calling this function:
  - The interrupt handler for the SSI module must call [SPIFlashIntHandler\(\)](#), passing the same pState structure pointer that was supplied to this function.
  - No other SPI flash operation can be called until this operation has completed.

Completion of the read operation is indicated when [SPIFlashIntHandler\(\)](#) returns **SPI\_FLASH\_DONE**.

Like [SPIFlashQuadRead\(\)](#), this uses the 0x6b SPI flash command.

#### Returns:

None.

### 18.2.2.14 SPIFlashRead

Reads data from the SPI flash.

#### Prototype:

```
void
SPIFlashRead(uint32_t ui32Base,
             uint32_t ui32Addr,
             uint8_t *pui8Data,
             uint32_t ui32Count)
```

#### Parameters:

**ui32Base** is the SSI module base address.

**ui32Addr** is the SPI flash address to read.

**pui8Data** is a pointer to the data buffer to into which to read the data.

**ui32Count** is the number of bytes to read.

**Description:**

This function reads data from the SPI flash, using PIO mode. This function will not return until the read has completed. This uses the 0x03 SPI flash command.

**Returns:**

None.

### 18.2.2.15 SPIFlashReadID

Reads the manufacturer and device IDs from the SPI flash.

**Prototype:**

```
void  
SPIFlashReadID(uint32_t ui32Base,  
               uint8_t *pui8ManufacturerID,  
               uint16_t *pui16DeviceID)
```

**Parameters:**

***ui32Base*** is the SSI module base address.

***pui8ManufacturerID*** is a pointer to the location into which to store the manufacturer ID.

***pui16DeviceID*** is a pointer to the location into which to store the device ID.

**Description:**

This function reads the manufacturer and device IDs from the SPI flash. These values can be used to identify the SPI flash that is attached, as well as determining if a SPI flash is attached (if the **SSIRx** pin is pulled up or down, either using the pad's weak pull up/down or using an external resistor, which will cause the returned IDs to be either all zeros or all ones if the SPI flash is not attached). This uses the 0x9f SPI flash command.

**Returns:**

None.

### 18.2.2.16 SPIFlashReadNonBlocking

Reads data from the SPI flash in the background.

**Prototype:**

```
void  
SPIFlashReadNonBlocking(tSPIFlashState *pState,  
                        uint32_t ui32Base,  
                        uint32_t ui32Addr,  
                        uint8_t *pui8Data,  
                        uint32_t ui32Count,  
                        bool bUseDMA,  
                        uint32_t ui32TxChannel,  
                        uint32_t ui32RxChannel)
```

**Parameters:**

***pState*** is a pointer to the SPI flash state structure.

***ui32Base*** is the SSI module base address.



***ui32Addr*** is the SPI flash address to read.

***pui8Data*** is a pointer to the data buffer to into which to read the data.

***ui32Count*** is the number of bytes to read.

***bUseDMA*** is **true** if uDMA should be used and **false** otherwise.

***ui32TxChannel*** is the uDMA channel to be used for writing to the SSI module.

***ui32RxChannel*** is the uDMA channel to be used for reading from the SSI module.

**Description:**

This function reads data from the SPI flash, using either interrupts or uDMA to transfer the data. This function will return immediately and read the data in the background. In order for this to complete successfully, several conditions must be satisfied:

- Prior to calling this function:
  - The SSI module must be enabled in SysCtl.
  - The SSI pins must be configured for use by the SSI module.
  - The SSI module interrupt must be enabled in NVIC.
  - The uDMA module must be enabled in SysCtl and the control table set (if using uDMA).
  - The uDMA channels must be assigned to the SSI module.
- After calling this function:
  - The interrupt handler for the SSI module must call [SPIFlashIntHandler\(\)](#), passing the same pState structure pointer that was supplied to this function.
  - No other SPI flash operation can be called until this operation has completed.

Completion of the read operation is indicated when [SPIFlashIntHandler\(\)](#) returns **SPI\_FLASH\_DONE**.

Like [SPIFlashRead\(\)](#), this uses the 0x03 SPI flash command.

**Returns:**

None.

### 18.2.2.17 SPIFlashReadStatus

Reads the SPI flash status register.

**Prototype:**

```
uint8_t  
SPIFlashReadStatus(uint32_t ui32Base)
```

**Parameters:**

***ui32Base*** is the SSI module base address.

**Description:**

This function reads the SPI flash status register. This uses the 0x05 SPI flash command.

**Returns:**

Returns the value of the SPI flash status register.

### 18.2.2.18 SPIFlashSectorErase

Erases a 4 KB sector of the SPI flash.

**Prototype:**

```
void  
SPIFlashSectorErase (uint32_t ui32Base,  
                     uint32_t ui32Addr)
```

**Parameters:**

***ui32Base*** is the SSI module base address.

***ui32Addr*** is the SPI flash address to erase.

**Description:**

This function erases a sector of the SPI flash. Each sector is 4 KB with a 4 KB alignment; the SPI flash will ignore the lower ten bits of the address provided. The sector erase command is issued by this function; [SPIFlashReadStatus\(\)](#) must be used to query the SPI flash to determine when the sector erase operation has completed. This uses the 0x20 SPI flash command.

**Returns:**

None.

### 18.2.2.19 SPIFlashWriteDisable

Disables SPI flash write operations.

**Prototype:**

```
void  
SPIFlashWriteDisable (uint32_t ui32Base)
```

**Parameters:**

***ui32Base*** is the SSI module base address.

**Description:**

This function sets the SPI flash to disallow program and erase operations. This uses the 0x04 SPI flash command.

**Returns:**

None.

### 18.2.2.20 SPIFlashWriteEnable

Enables SPI flash write operations.

**Prototype:**

```
void  
SPIFlashWriteEnable (uint32_t ui32Base)
```

**Parameters:**

***ui32Base*** is the SSI module base address.

**Description:**

This function sets the SPI flash to allow program and erase operations. This must be done prior to each SPI flash program or erase operation; the SPI flash will automatically disable program and erase operations once a program or erase operation has completed. This uses the 0x06 SPI flash command.

**Returns:**

None.

### 18.2.2.21 SPIFlashWriteStatus

Writes the SPI flash status register.

**Prototype:**

```
void  
SPIFlashWriteStatus(uint32_t ui32Base,  
                    uint8_t ui8Status)
```

**Parameters:**

***ui32Base*** is the SSI module base address.

***ui8Status*** is the value to write to the status register.

**Description:**

This function writes the SPI flash status register. This uses the 0x01 SPI flash command.

**Returns:**

None.

## 18.3 Programming Example

The following example shows how to produce a sine wave with 7 degrees between successive values.

```
// TODO: Add this
```



# 19 Ethernet Software Update Module

Introduction .....	165
API Functions .....	165
Programming Example .....	167

## 19.1 Introduction

The Ethernet software update module provides a convenient method of registering a callback which will be notified when a user attempts to initiate a firmware update over Ethernet using the LM Flash Programmer application. In addition to providing notification of an update request, the module also provides a function that can be called to initiate an update using the Ethernet boot loader.

To make use of this module, an application must include the lwIP TCP/IP stack and must be run on a system configured to use the Ethernet boot loader.

This module is contained in `utils/swupdate.c`, with `utils/swupdate.h` containing the API declarations for use by applications.

## 19.2 API Functions

### Functions

- void [SoftwareUpdateBegin](#) (void)
- void [SoftwareUpdateInit](#) (tSoftwareUpdateRequested pfnCallback)

### 19.2.1 Function Documentation

#### 19.2.1.1 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.

### 19.2.1.2 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-signaled 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 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 signaling.

**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.

## 19.3 Programming Example

The following example shows how to use the software update module.

```
//*****
//
// A flag used to indicate that an Ethernet remote firmware update request
// has been received.
//
//*****
volatile tBoolean g_bFirmwareUpdate = false;

//*****
//
// This function is called by the software update module whenever a remote
// host requests to update the firmware on this board. We set a flag that
// will cause the bootloader to be entered the next time the user enters a
// command on the console.
//
//*****
void
SoftwareUpdateRequestCallback(void)
{
    g_bFirmwareUpdate = true;
}

//*****
//
// The main entry point for the application. This function contains all
// hardware initialization code and also the main loop for the application.
//
//*****
int
main(void)
{
    unsigned char pucMACAddr[6];

    //
    // System clock initialization and reading of the MAC address into array
    // pucMACAddr occurs here. This code is omitted for clarity.
    //

    //
    // Initialize the lwIP TCP/IP stack.
    //
    lwIPInit(pucMACAddr, 0, 0, 0, IPADDR_USE_DHCP);

    //
    // Start the remote software update module.
    //
    SoftwareUpdateInit(SoftwareUpdateRequestCallback);

    //
    // Do whatever other setup things the application needs.
    //

    //
    // Loop until someone requests a remote firmware update.
    //
    while(!g_bFirmwareUpdate)
    {
        //
        // Perform your main loop functions here.
        //
    }
}
```

```
    }  
  
    //  
    // If we drop out, a remote firmware update request has been received.  
    // Transfer control to the bootloader which will perform the update.  
    //  
    SoftwareUpdateBegin();  
}
```



## 20 UART Standard IO Module

Introduction .....	169
API Functions .....	170
Programming Example .....	176

### 20.1 Introduction

The UART standard IO module provides a simple interface to a UART that is similar to the standard IO package available in the C library. Only a very small subset of the normal functions are provided; [UARTprintf\(\)](#) is an equivalent to the C library `printf()` function and [UARTgets\(\)](#) is an equivalent to the C library `fgets()` function.

This module is contained in `utils/uartstdio.c`, with `utils/uartstdio.h` containing the API declarations for use by applications.

#### 20.1.1 Unbuffered Operation

Unbuffered operation is selected by not defining **UART\_BUFFERED** when building the UART standard IO module. In unbuffered mode, calls to the module will not return until the operation has been completed. So, for example, a call to [UARTprintf\(\)](#) will not return until the entire string has been placed into the UART's FIFO. If it is not possible for the function to complete its operation immediately, it will busy wait.

#### 20.1.2 Buffered Operation

Buffered operation is selected by defining **UART\_BUFFERED** when building the UART standard IO module. In buffered mode, there is a larger UART data FIFO in SRAM that extends the size of the hardware FIFO. Interrupts from the UART are used to transfer data between the SRAM buffer and the hardware FIFO. It is the responsibility of the application to ensure that [UARTStdioIntHandler\(\)](#) is called when the UART interrupt occurs; typically this is accomplished by placing it in the vector table in the startup code for the application.

In addition to providing a larger UART buffer, the behavior of [UARTprintf\(\)](#) is slightly modified. If the output buffer is full, [UARTprintf\(\)](#) will discard the remaining characters from the string instead of waiting until space becomes available in the buffer. If this behavior is not desired, [UARTFlushTx\(\)](#) may be called to ensure that the transmit buffer is emptied prior to adding new data via [UARTprintf\(\)](#) (though this will not work if the string to be printed is larger than the buffer).

[UARTPeek\(\)](#) can be used to determine whether a line end is present prior to calling [UARTgets\(\)](#) if a non-blocking operation is required. In cases where the buffer supplied on [UARTgets\(\)](#) fills before a line termination character is received, the call will return with a full buffer.

## 20.2 API Functions

### Functions

- void [UARTEchoSet](#) (bool bEnable)
- void [UARTFlushRx](#) (void)
- void [UARTFlushTx](#) (bool bDiscard)
- unsigned char [UARTgetc](#) (void)
- int [UARTgets](#) (char \*pcBuf, uint32\_t ui32Len)
- int [UARTPeek](#) (unsigned char ucChar)
- void [UARTprintf](#) (const char \*pcString, ...)
- int [UARTRxBytesAvail](#) (void)
- void [UARTStdioConfig](#) (uint32\_t ui32PortNum, uint32\_t ui32Baud, uint32\_t ui32SrcClock)
- void [UARTStdioIntHandler](#) (void)
- int [UARTTxBytesFree](#) (void)
- void [UARTvprintf](#) (const char \*pcString, va\_list vaArgP)
- int [UARTwrite](#) (const char \*pcBuf, uint32\_t ui32Len)

### 20.2.1 Function Documentation

#### 20.2.1.1 UARTEchoSet

Enables or disables echoing of received characters to the transmitter.

**Prototype:**

```
void
UARTEchoSet (bool bEnable)
```

**Parameters:**

**bEnable** must be set to **true** to enable echo or **false** to disable it.

**Description:**

This function, available only when the module is built to operate in buffered mode using **UART\_BUFFERED**, may be used to control whether or not received characters are automatically echoed back to the transmitter. By default, echo is enabled and this is typically the desired behavior if the module is being used to support a serial command line. In applications where this module is being used to provide a convenient, buffered serial interface over which application-specific binary protocols are being run, however, echo may be undesirable and this function can be used to disable it.

**Returns:**

None.

#### 20.2.1.2 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\(\)](#).

**Returns:**

None.

### 20.2.1.3 UARTFlushTx

Flushes the transmit buffer.

**Prototype:**

```
void
UARTFlushTx(bool 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 is waiting to be transmitted. On return, the transmit buffer will be empty.

**Returns:**

None.

### 20.2.1.4 UARTgetc

Read a single character from the UART, blocking if necessary.

**Prototype:**

```
unsigned char
UARTgetc(void)
```

**Description:**

This function will receive a single character from the UART and store it at the supplied address.

In both buffered and unbuffered modes, this function will block until a character is received. If non-blocking operation is required in buffered mode, a call to [UARTRxAvail\(\)](#) may be made to determine whether any characters are currently available for reading.

**Returns:**

Returns the character read.

### 20.2.1.5 UARTgets

A simple UART based get string function, with some line processing.

**Prototype:**

```
int
UARTgets(char *pcBuf,
          uint32_t ui32Len)
```

**Parameters:**

**pcBuf** points to a buffer for the incoming string from the UART.

**ui32Len** 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.

**Returns:**

Returns the count of characters that were stored, not including the trailing 0.

### 20.2.1.6 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.

**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.

### 20.2.1.7 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 or %i 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 upper case letters (not lower case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %s, %d, %i, %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 preceded 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.

**Returns:**

None.

### 20.2.1.8 UARTRxBytesAvail

Returns the number of bytes available in the receive buffer.

**Prototype:**

```
int
UARTRxBytesAvail(void)
```

**Description:**

This function, available only when the module is built to operate in buffered mode using **UART\_BUFFERED**, may be used to determine the number of bytes of data currently available in the receive buffer.

**Returns:**

Returns the number of available bytes.

### 20.2.1.9 UARTStdioConfig

Configures the UART console.

**Prototype:**

```
void
UARTStdioConfig(uint32_t ui32PortNum,
                 uint32_t ui32Baud,
                 uint32_t ui32SrcClock)
```

**Parameters:**

***ui32PortNum*** is the number of UART port to use for the serial console (0-2)

***ui32Baud*** is the bit rate that the UART is to be configured to use.

***ui32SrcClock*** is the frequency of the source clock for the UART module.

**Description:**

This function will configure the specified serial port to be used as a serial console. The serial parameters are set to the baud rate specified by the *ui32Baud* parameter and use 8 bit, no parity, and 1 stop bit.

This function must be called prior to using any of the other UART console functions: [UARTprintf\(\)](#) or [UARTgets\(\)](#). This function assumes that the caller has previously configured the relevant UART pins for operation as a UART rather than as GPIOs.

**Returns:**

None.

### 20.2.1.10 UARTStdioIntHandler

Handles UART interrupts.

**Prototype:**

```
void
UARTStdioIntHandler(void)
```

**Description:**

This function handles interrupts from the UART. It will copy data from the transmit buffer to the UART transmit FIFO if space is available, and it will copy data from the UART receive FIFO to the receive buffer if data is available.

**Returns:**

None.

### 20.2.1.11 UARTTxBytesFree

Returns the number of bytes free in the transmit buffer.

**Prototype:**

```
int
UARTTxBytesFree(void)
```

**Description:**

This function, available only when the module is built to operate in buffered mode using **UART\_BUFFERED**, may be used to determine the amount of space currently available in the transmit buffer.

**Returns:**

Returns the number of free bytes.

### 20.2.1.12 UARTvprintf

A simple UART based vprintf function supporting %c, %d, %p, %s, %u, %x, and %X.

**Prototype:**

```
void
UARTvprintf(const char *pcString,
            va_list vaArgP)
```

**Parameters:**

**pcString** is the format string.

**vaArgP** is a variable argument list pointer whose content will depend upon the format string passed in *pcString*.

**Description:**

This function is very similar to the C library `vprintf()` function. All of its output will be sent to the UART. Only the following formatting characters are supported:

- %c to print a character
- %d or %i 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 upper case letters (not lower case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %s, %d, %i, %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 preceded 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 in the variable arguments list 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.

**Returns:**

None.

### 20.2.1.13 UARTwrite

Writes a string of characters to the UART output.

**Prototype:**

```
int
UARTwrite(const char *pcBuf,
          uint32_t ui32Len)
```

**Parameters:**

**pcBuf** points to a buffer containing the string to transmit.

**ui32Len** is the length of the string to transmit.

**Description:**

This function will transmit the string to the UART output. The number of characters transmitted is determined by the *ui32Len* parameter. This function does no interpretation or translation of any characters. Since the output is sent to a UART, any LF (/n) characters encountered will be replaced with a CRLF pair.

Besides using the *ui32Len* parameter to stop transmitting the string, if a null character (0) is encountered, then no more characters will be transmitted and the function will return.

In non-buffered mode, this function is blocking and will not return until all the characters have been written to the output FIFO. In buffered mode, the characters are written to the UART transmit buffer and the call returns immediately. If insufficient space remains in the transmit buffer, additional characters are discarded.

**Returns:**

Returns the count of characters written.

## 20.3 Programming Example

The following example shows how to use the UART standard IO module to write a string to the UART “console”.

```
//
// Configure the appropriate pins as UART pins; in this case, PA0/PA1 are
// used for UART0.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

//
// Initialize the UART standard IO module, when using an 80 MHz system
// clock.
//
UARTStdioConfig(0, 115200, 80000000);

//
// Print a string.
//
UARTprintf("Hello world!\n");
```



## 21 Micro Standard Library Module

Introduction .....	177
API Functions .....	177
Programming Example .....	186

### 21.1 Introduction

The micro standard library module provides a set of small implementations of functions normally found in the C library. These functions provide reduced or greatly reduced functionality in order to remain small while still being useful for most embedded applications.

The following functions are provided, along with the C library equivalent:

Function	C library equivalent
<code>ulocaltime</code>	<code>localtime</code>
<code>umktime</code>	<code>mktime</code>
<code>urand</code>	<code>rand</code>
<code>usnprintf</code>	<code>snprintf</code>
<code>usprintf</code>	<code>sprintf</code>
<code>usrand</code>	<code>srand</code>
<code>ustrcasecmp</code>	<code>strcasecmp</code>
<code>strcmp</code>	<code>strcmp</code>
<code>strlen</code>	<code>strlen</code>
<code>strncmp</code>	<code>strncmp</code>
<code>strncpy</code>	<code>strncpy</code>
<code>strnicmp</code>	<code>strnicmp</code>
<code>strstr</code>	<code>strstr</code>
<code>strtouf</code>	<code>strtouf</code>
<code>strtoul</code>	<code>strtoul</code>
<code>uvsnprintf</code>	<code>vsnprintf</code>

This module is contained in `utils/ustdlib.c`, with `utils/ustdlib.h` containing the API declarations for use by applications.

### 21.2 API Functions

#### Functions

- void `ulocaltime` (time\_t timer, struct tm \*tm)
- time\_t `umktime` (struct tm \*timeptr)
- int `urand` (void)
- int `usnprintf` (char \*restrict s, size\_t n, const char \*restrict format, ...)
- int `usprintf` (char \*restrict s, const char \*format, ...)
- void `usrand` (unsigned int seed)
- int `ustrcasecmp` (const char \*s1, const char \*s2)

- int [ustrcmp](#) (const char \*s1, const char \*s2)
- size\_t [ustrlen](#) (const char \*s)
- int [ustrncasecmp](#) (const char \*s1, const char \*s2, size\_t n)
- int [ustrncmp](#) (const char \*s1, const char \*s2, size\_t n)
- char \* [ustrncpy](#) (char \*restrict s1, const char \*restrict s2, size\_t n)
- char \* [ustrstr](#) (const char \*s1, const char \*s2)
- float [ustrtof](#) (const char \*nptr, const char \*\*endptr)
- unsigned long [ustrtoul](#) (const char \*restrict nptr, const char \*\*restrict endptr, int base)
- int [uvsnprintf](#) (char \*restrict s, size\_t n, const char \*restrict format, va\_list arg)

## 21.2.1 Function Documentation

### 21.2.1.1 `ulocaltime`

Converts from seconds to calendar date and time.

**Prototype:**

```
void  
ulocaltime(time_t timer,  
            struct tm *tm)
```

**Parameters:**

***timer*** is the number of seconds.

***tm*** 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.

### 21.2.1.2 `umktime`

Converts calendar date and time to seconds.

**Prototype:**

```
time_t  
umktime(struct tm *timeptr)
```

**Parameters:**

***timeptr*** is a pointer to the time structure that is filled in with the broken down date and time.

**Description:**

This function converts the date and time represented by the *timeptr* structure pointer to the number of seconds since midnight GMT on January 1, 1970 (traditional Unix epoch).

**Returns:**

Returns the calendar time and date as seconds. If the conversion was not possible then the function returns (uint32\_t)(-1).

### 21.2.1.3 urand

Generate a new (pseudo) random number

**Prototype:**

```
int
urand(void)
```

**Description:**

This function is very similar to the C library `rand()` function. It will generate a pseudo-random number sequence based on the seed value.

**Returns:**

A pseudo-random number will be returned.

### 21.2.1.4 usnprintf

A simple `snprintf` function supporting %c, %d, %p, %s, %u, %x, and %X.

**Prototype:**

```
int
usnprintf(char *restrict s,
          size_t n,
          const char *restrict format,
          ...)
```

**Parameters:**

**s** is the buffer where the converted string is stored.

**n** is the size of the buffer.

**format** 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 or %i 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 upper case letters (not lower case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %d, %i, %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 preceded 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 *format* 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  $n - 1$  characters into the buffer *s*. 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.

### 21.2.1.5 `usprintf`

A simple `sprintf` function supporting `%c`, `%d`, `%p`, `%s`, `%u`, `%x`, and `%X`.

**Prototype:**

```
int
usprintf(char *restrict s,
         const char *format,
         ...)
```

**Parameters:**

***s*** is the buffer where the converted string is stored.

***format*** 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` or `%i` 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 upper case letters (not lower case letters as would typically be used)
- `%p` to print a pointer as a hexadecimal value
- `%%` to print out a `%` character

For `%d`, `%i`, `%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 preceded 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 *format* 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 *s* 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.

### 21.2.1.6 `usrand`

Set the random number generator seed.

**Prototype:**

```
void  
usrand(unsigned int seed)
```

**Parameters:**

***seed*** is the new seed value to use for the random number generator.

**Description:**

This function is very similar to the C library `srand()` function. It will set the seed value used in the `urand()` function.

**Returns:**

None

### 21.2.1.7 `ustrcasecmp`

Compares two strings without regard to case.

**Prototype:**

```
int  
ustrcasecmp(const char *s1,  
             const char *s2)
```

**Parameters:**

***s1*** points to the first string to be compared.

***s2*** points to the second string to be compared.

**Description:**

This function is very similar to the C library `strcasecmp()` function. It compares two strings without regard to case. The comparison ends if a terminating NULL character is found in either string. In this case, the `int16_t` string is deemed the lesser.

**Returns:**

Returns 0 if the two strings are equal, -1 if *s1* is less than *s2* and 1 if *s1* is greater than *s2*.

### 21.2.1.8 `ustrcmp`

Compares two strings.

**Prototype:**

```
int
ustrcmp(const char *s1,
        const char *s2)
```

**Parameters:**

- s1** points to the first string to be compared.
- s2** points to the second string to be compared.

**Description:**

This function is very similar to the C library `strcmp()` function. It compares two strings, taking case into account. The comparison ends if a terminating NULL character is found in either string. In this case, the `int16_t` string is deemed the lesser.

**Returns:**

Returns 0 if the two strings are equal, -1 if *s1* is less than *s2* and 1 if *s1* is greater than *s2*.

### 21.2.1.9 `ustrlen`

Returns the length of a null-terminated string.

**Prototype:**

```
size_t
ustrlen(const char *s)
```

**Parameters:**

- s** is a pointer to the string whose length is to be found.

**Description:**

This function is very similar to the C library `strlen()` function. It determines the length of the null-terminated string passed and returns this to the caller.

This implementation assumes that single byte character strings are passed and will return incorrect values if passed some UTF-8 strings.

**Returns:**

Returns the length of the string pointed to by *s*.

### 21.2.1.10 `ustrncasecmp`

Compares two strings without regard to case.

**Prototype:**

```
int
ustrncasecmp(const char *s1,
             const char *s2,
             size_t n)
```

**Parameters:**

- s1** points to the first string to be compared.
- s2** points to the second string to be compared.
- n** is the maximum number of characters to compare.

**Description:**

This function is very similar to the C library `strncasecmp()` function. It compares at most *n* characters of two strings without regard to case. The comparison ends if a terminating NULL character is found in either string before *n* characters are compared. In this case, the shorter string is deemed the lesser.

**Returns:**

Returns 0 if the two strings are equal, -1 if *s1* is less than *s2* and 1 if *s1* is greater than *s2*.

### 21.2.1.11 `ustrncmp`

Compares two strings.

**Prototype:**

```
int
ustrncmp(const char *s1,
         const char *s2,
         size_t n)
```

**Parameters:**

- s1** points to the first string to be compared.
- s2** points to the second string to be compared.
- n** is the maximum number of characters to compare.

**Description:**

This function is very similar to the C library `strncmp()` function. It compares at most *n* characters of two strings taking case into account. The comparison ends if a terminating NULL character is found in either string before *n* characters are compared. In this case, the `int16_t` string is deemed the lesser.

**Returns:**

Returns 0 if the two strings are equal, -1 if *s1* is less than *s2* and 1 if *s1* is greater than *s2*.

### 21.2.1.12 `ustrncpy`

Copies a certain number of characters from one string to another.

**Prototype:**

```
char *
ustrncpy(char *restrict s1,
         const char *restrict s2,
         size_t n)
```

**Parameters:**

- s1** is a pointer to the destination buffer into which characters are to be copied.

**s2** is a pointer to the string from which characters are to be copied.  
**n** is the number of characters to copy to the destination buffer.

**Description:**

This function copies at most *n* characters from the string pointed to by *s2* into the buffer pointed to by *s1*. If the end of *s2* is found before *n* characters have been copied, remaining characters in *s1* will be padded with zeroes until *n* characters have been written. Note that the destination string will only be NULL terminated if the number of characters to be copied is greater than the length of *s2*.

**Returns:**

Returns *s1*.

### 21.2.1.13 ustrstr

Finds a substring within a string.

**Prototype:**

```
char *  
ustrstr(const char *s1,  
        const char *s2)
```

**Parameters:**

**s1** is a pointer to the string that will be searched.  
**s2** is a pointer to the substring that is to be found within *s1*.

**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 *s2* within *s1* or NULL if no match is found.

### 21.2.1.14 strttof

Converts a string into its floating-point equivalent.

**Prototype:**

```
float  
strttof(const char *nptr,  
        const char **endptr)
```

**Parameters:**

**nptr** is a pointer to the string containing the floating-point value.  
**endptr** is a pointer that will be set to the first character past the floating-point value in the string.

**Description:**

This function is very similar to the C library `strttof()` function. It scans a string for the first token (that is, non-white space) and converts the value at that location in the string into a floating-point value.



**Returns:**

Returns the result of the conversion.

### 21.2.1.15 strtoul

Converts a string into its numeric equivalent.

**Prototype:**

```
unsigned long
strtoul(const char *restrict nptr,
        const char **restrict endptr,
        int base)
```

**Parameters:**

***nptr*** is a pointer to the string containing the integer.

***endptr*** is a pointer that will be set to the first character past the integer in the string.

***base*** 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-white space) and converts the value at that location in the string into an integer value.

**Returns:**

Returns the result of the conversion.

### 21.2.1.16 uvsnprintf

A simple `vsprintf` function supporting `%c`, `%d`, `%p`, `%s`, `%u`, `%x`, and `%X`.

**Prototype:**

```
int
uvsnprintf(char *restrict s,
           size_t n,
           const char *restrict format,
           va_list arg)
```

**Parameters:**

***s*** points to the buffer where the converted string is stored.

***n*** is the size of the buffer.

***format*** is the format string.

***arg*** 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 `vsprintf()` function. Only the following formatting characters are supported:

- `%c` to print a character
- `%d` or `%i` 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 upper case letters (not lower case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %d, %i, %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 preceded 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 *format* 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 *n* parameter limits the number of characters that will be stored in the buffer pointed to by *s* 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.

## 21.3 Programming Example

The following example shows how to use some of the micro standard library functions.

```
unsigned long ulValue;
char pcBuffer[32];
tTime sTime;

//
// Convert the number in pcBuffer (previous read from somewhere) into an
// integer. Note that this supports converting decimal values (such as
// 4583), octal values (such as 036573), and hexadecimal values (such as
// 0x3425).
//
ulValue = strtoul(pcBuffer, 0, 0);

//
// Convert that integer from a number of seconds into a broken down date.
//
ulocaltime(ulValue, &sTime);

//
// Print out the corresponding time of day in military format.
//
usprintf(pcBuffer, "%02d:%02d", sTime.ucHour, sTime.ucMin);
```



---

# IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have **not** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

## Products

Audio	<a href="http://www.ti.com/audio">www.ti.com/audio</a>
Amplifiers	<a href="http://amplifier.ti.com">amplifier.ti.com</a>
Data Converters	<a href="http://dataconverter.ti.com">dataconverter.ti.com</a>
DLP® Products	<a href="http://www.dlp.com">www.dlp.com</a>
DSP	<a href="http://dsp.ti.com">dsp.ti.com</a>
Clocks and Timers	<a href="http://www.ti.com/clocks">www.ti.com/clocks</a>
Interface	<a href="http://interface.ti.com">interface.ti.com</a>
Logic	<a href="http://logic.ti.com">logic.ti.com</a>
Power Mgmt	<a href="http://power.ti.com">power.ti.com</a>
Microcontrollers	<a href="http://microcontroller.ti.com">microcontroller.ti.com</a>
RFID	<a href="http://www.ti-rfid.com">www.ti-rfid.com</a>
OMAP Applications Processors	<a href="http://www.ti.com/omap">www.ti.com/omap</a>
Wireless Connectivity	<a href="http://www.ti.com/wirelessconnectivity">www.ti.com/wirelessconnectivity</a>

## Applications

Automotive and Transportation	<a href="http://www.ti.com/automotive">www.ti.com/automotive</a>
Communications and Telecom	<a href="http://www.ti.com/communications">www.ti.com/communications</a>
Computers and Peripherals	<a href="http://www.ti.com/computers">www.ti.com/computers</a>
Consumer Electronics	<a href="http://www.ti.com/consumer-apps">www.ti.com/consumer-apps</a>
Energy and Lighting	<a href="http://www.ti.com/energy">www.ti.com/energy</a>
Industrial	<a href="http://www.ti.com/industrial">www.ti.com/industrial</a>
Medical	<a href="http://www.ti.com/medical">www.ti.com/medical</a>
Security	<a href="http://www.ti.com/security">www.ti.com/security</a>
Space, Avionics and Defense	<a href="http://www.ti.com/space-avionics-defense">www.ti.com/space-avionics-defense</a>
Video and Imaging	<a href="http://www.ti.com/video">www.ti.com/video</a>

## TI E2E Community

[e2e.ti.com](http://e2e.ti.com)

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265  
Copyright © 2013, Texas Instruments Incorporated