Connectivity Processor Driver Library

User's Manual

Revision 1.5

National Semiconductor
The Sight & Sound of Information

Preface

Scope of This Document

This document is a guide and reference manual for the peripheral driver library provided for the Connectivity Processors (CP3BT1x/CP3CN1x/CP3UP1x).

Related Documentation

CompactRISC CR16C Programmer's Reference Manual—This is the authoritative reference for the architecture of the CR16C CPU. Compiler writers and assembly-language programmers should consult this document for detailed information about the instruction set.

Device Datasheets—Refer to the datasheets for the individual Connectivity Processor device types for detailed information about the on-chip peripheral devices, signal descriptions, package pinout, and electrical specifications.

Preface

Notational Conventions

Commands selected from menus are shown as "File -> New", which represents the "New" command selected from the "File" menu.

High-level language and assembly code, command lines, and macro file statements are shown in the Courier font, for example:

```
add r2,r3;
```

When a single command or statement is too long to fit on one line, a backslash (\) is used to indicate continuation on the following line, for example:

```
echo "ERROR 137 -- square root domain error, \
try using a positive number"
```

Revision History

Revision	Release Date	Summary of Changes
1.5	9/8/03	Original release.

Preface

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1 Overview

The Connectivity Processors include a rich set of on-chip I/O peripherals for supporting a broad range of embedded applications. To help application developers use these peripherals, a set of software drivers is provided for accessing the peripherals through generic software APIs (Application Programming Interfaces). For most applications, these APIs eliminate the need to have an intimate knowledge of the register-level interface to the peripherals. Source code for the drivers is available, to allow customizing the drivers for specific needs.

This document describes:

Chapter 2—Driver Library. Describes the contents of the driver library and how to integrate the library into applications.

Chapter 3—Accessing I/O Registers. Describes how to access peripheral I/O registers from application code written in C.

Chapter 4—Data Link Mechanism. Describes the data link mechanism for streaming data between the application and peripherals.

Chapter 5—Interrupts. Describes interrupt service routines and the interrupt table.

Appendix A—Driver Library API. Defines the function calls to the driver library API.

Overview

Using the Driver Library 2

2.1 Driver Library Header Files

Driver libraries are provided for both the on-chip peripherals and on-board peripherals (external to the CP3000 device).

Application programs that use the drivers for on-chip peripherals must include the **drivers.h** header file, located in the **include** directory. The **drivers.h** file contains function prototypes for the API functions and additional definitions such as data structures and other types used by the on-chip peripheral drivers.

Application programs that use the board-level peripheral drivers must include the **boarddrv.h** header file, also located in the **include** directory. The **boarddrv.h** file contains prototypes and definitions used by drivers for external peripherals, such as LEDs, DIP switches, EEPROM, etc.

2.2 Library Contents

The modules in the driver library are listed in Table 2-1.

Table 2-1. Modules in the Connectivity Processor Driver Library

File Name	Module
`acbm.c`	ACCESS.bus (master mode)
`acbs.c`	ACCESS.bus (slave mode)
`adc.c`	Analog/Digital Converter
`audio.c`	Advanced Audio Interface
`biu.c`	Bus Interface Unit
`btllc.c`	Bluetooth Lower Link Controller
`can.c`	Controller Area Network
`flash.c`	Flash Memory
`icu.c`	Interrupt Control Unit
`lmx5250.c`	LMX5250 (radio chip)
`mft.c`	Multi Function Timer
`mw.c`	Microwire
`syscfg.c`	System Configuration
`tcr.c`	Triple Clock and Reset
`timer.c`	System Timer (currently uses Timing and Watchdog Module)
`twm.c`	Timing and Watchdog Module
`usart.c`	UART (Interrupt mode)
`usart_dma.c`	UART (DMA mode)
`vtu.c`	Versatile Timer Unit

Accessing I/O Registers

3

Each of the drivers has to access some I/O registers (also known as Special Function Registers or SFRs) that are associated with I/O peripherals. In the driver source code, each register is referred to by its name as defined in the device datasheet.

The actual implementation of I/O registers in the driver library uses one structure for each I/O peripheral registers set. These structures are defined in the **cp3bt1x.h** header file. Each structure member represents a register or a space between registers (spaces exist because I/O registers are always assigned an even address -- if a register is one byte long there a one-byte space between the register and the following register). Each structure represents the memory layout of register set associated with one peripheral. An example of a peripheral register set structure is:

```
typedef volatile FARIO struct {
  unsigned char bcfg;
  unsigned char f1;
  unsigned short iocfg;
  unsigned short szcfg0;
  unsigned short szcfg1;
  unsigned short szcfg2;
} biu_t;
```

Accessing I/O Registers

For each peripheral, a constant structure pointer is defined which points to the base address of the peripheral register set. To access a register, access the structure member through the constant pointer. For example, the pointer to the BIU register set is defined as follows:

```
biu_t * const biu = (biu_t *)BIU_BASE_ADDR;
```

The IOCFG register of the BIU peripheral is accessed as follows:

```
biu->iocfg
```

Finally, there is a definition for each register that allows using only its name for simplicity and readability:

```
#define IOCFG biu->iocfg
```

In this case, the IOCFG register of the BIU peripheral is referred to in the driver code simply as **`IOCFG`**.

Data Link Mechanism

Some drivers stream data between the application and the peripheral using the *data link* mechanism.

The data link mechanism is a simple concept used to synchronize a producer and a consumer. A producer continuously generates data that is later read by the consumer. They are not synchronized except through the data link mechanism. The data link is implemented as a circular buffer and a control structure that keeps all the necessary information about the status of the buffer. The circular buffer is used for the data transfer between the producer and the consumer.

The most important field in the data link structure is the data link counter, which indicates how many bytes have been written to the buffer by the producer and not yet read from the buffer by the consumer. The value in this counter can be anywhere between zero and the size of the circular buffer in bytes. The consumer and the producer each have a private pointer to the circular buffer. The producer pointer points to the location in the buffer to which the producer should write the next byte. The consumer pointer points to the location in the buffer from which the consumer will read the next byte. The producer can write to the data link buffer as long as the buffer is not full (data link counter less than buffer size). The consumer can read from the data link buffer as long as the buffer is not empty (data link counter not equal to zero).

Data Link Mechanism

The data link structure and a set of useful data link macros are defined in the **dl.h** header file under the **include** directory. Some of the important primitives provided by these macros are described in Table 4-1.

Table 4-1. Data Link Primitives

Name	Description
`dl_advance`	Advances a pointer by one position in the circular buffer. Performs a wrap around if required.
`dl_p_commit`	Producer commit. The producer releases a given number of bytes to the consumer. The bytes have already been written and now they are confirmed as released by incrementing the data link counter.
`dl_c_commit`	Consumer commit. The consumer reports unloading a certain amount of bytes. The bytes have already been read and now they are confirmed as unloaded by decrementing the data link counter. The memory space previously occupied by these bytes is now available for writing by the producer.
`dl_empty`	A predicate telling whether the data link buffer is empty or not.
`dl_full`	A predicate telling whether the data link buffer is full or not.
`dl_cnt`	Provides the current number of bytes that have been written by the producer but not read by the consumer.
`dl_p_space_available`	Tells the producer how much space is available in the buffer for writing new data.

Interrupts

5

Many peripheral drivers must handle interrupts generated by their associated peripheral. These drivers include an interrupt handler function also called an *interrupt service routine* or *ISR*. Some drivers may include more than ISR, because they must handle more than one type of interrupt. An ISR is invoked by the CPU when the associated interrupt is triggered and several other conditions exist: the interrupt must be unmasked inside the interrupt control unit (ICU), interrupts must be globally enabled, and no other higher-priority interrupt may be pending.

The CPU uses an interrupt dispatch table to assign an ISR entry point address for each type of the interrupt. Each type of interrupt is assigned a number (also called vector). This number is used to calculate an offset into the interrupt dispatch table, from which the entry point address to the ISR is retrieved.

The interrupt table is generated automatically by the compiler. Each ISR is preceded by a definition of its vector number, and the compiler uses this information to generate an entry in the interrupt table. The compiler fills only entries for ISRs that are used in the application.

Interrupts

On-Chip Peripheral Drivers

A

A.1 Summary of On-Chip Peripheral Driver Functions

Table A-1 lists the on-chip peripheral driver functions.

Table A-1. On-Chip Peripheral Driver Functions

Module	Function Call	Section	Page	Description
ACCESS. bus (Master mode)	`acbm_init()`	A.2.1	33	Initializes the ACCESS.bus interface.
	`acbm_tx()`	A.2.2	34	Transmits data to specified slave device.
	`acbm_rx()`	A.2.3	35	Receives data from specified slave device.
ACCESS. bus (Slave mode)	`acbs_init()`	A.3.1	36	Initializes the ACCESS.bus interface.
	`acbs_tx()`	A.3.2	37	Deposits data to be transmitted on next master transmit request.
	`acbs_rx()`	A.3.3	38	Reads received bytes to a specified memory location.
	`acbs_rx_count()`	A.3.4	39	Returns number of bytes available for reading.

Table A-1. On-Chip Peripheral Driver Functions (Continued)

Module	Function Call	Section	Page	Description
Analog / Digital Converter	`AdcInit()`	A.4.1	40	Initializes ADC peripheral and driver.
	`AdcShutDown()`	A.4.2	42	Shuts down the ADC.
	`AdcTouchScreenConfig()`	A.4.3	42	Sets up ADC for reading touchscreen coordinates.
	`AdcStartConversion()`	A.4.4	43	Starts a conversion.
	`AdcReadResult()`	A.4.5	43	Reads the conversion result.
Advanced Audio Interface (AAI)	`audio_init()`	A.5.1	44	Initializes the AAI and CVSD / PCM hardware and driver.
	`audio_set_control_bits()`	A.5.2	45	Specifies control bit settings appended to each transmitted word.
	`audio_start()`	A.5.3	45	Starts the AAI and CVSD / PCM hardware.
	`audio_stop()`	A.5.4	45	Stops the AAI and CVSD / PCM hardware.
Bus Interface Unit (BIU)	`biu_config_zone()`	A.6.1	46	Configures a given memory or I / O zone according to specified parameters.
	`biu_set_early_write()`	A.6.2	47	Reports whether the BIU is in early-write or late-write mode.

Table A-1. On-Chip Peripheral Driver Functions (Continued)

Module	Function Call	Section	Page	Description
Controller Area Network (CAN)	`can_init()`	A.7.1	48	Initializes CAN hardware and driver.
	`can_config_tx()`	A.7.2	50	Configures a message buffer to transmit a packet.
	`can_config_rx()`	A.7.3	51	Configures the message buffer to receive a Data Frame only.
	`can_read_msg()`	A.7.4	52	Reads a message buffer.
	`can_shutdown()`	A.7.5	53	Shuts down the CAN driver.

Table A-1. On-Chip Peripheral Driver Functions (Continued)

Module	Function Call	Section	Page	Description
Flash Memory Interface	`FlashInit()`	A.8.1	54	Initializes the timing registers for the specified flash memory.
	`FlashGetPageSize()`	A.8.2	55	Returns the page size for specified flash memory.
	`FlashGetPageNum()`	A.8.3	55	Returns number of pages in specified flash memory.
	`FlashGetStartAddress()`	A.8.4	56	Returns start address for specified flash memory.
	`FlashInformationBlockRead()`	A.8.5	57	Reads a specified number of words from an Information Block.
	`FlashInformationBlockWrite()`	A.8.6	58	Writes a specified number of words to an Information Block
	`FlashInformationBlockErase()`	A.8.7	59	Erases an Information Block and corresponding Main Block.
	`FlashRead()`	A.8.8	60	Read specified number of bytes from Main Block.
	`FlashPageProgramSafe()`	A.8.9	61	Program specified number of bytes to Main Block.
	`FlashPageProgram()`	A.8.10	62	Program specfied page in Main Block.
	`FlashEraseAll()`	A.8.11	63	Erase specified flash memory.
	`FlashPageErase)`	A.8.12	64	Erase specified page.

Table A-1. On-Chip Peripheral Driver Functions (Continued)

Module	Function Call	Section	Page	Description
Interrupt Control Unit (ICU)	`icu_init()`	A.9.1	65	Initializes the ICU hardware.
	`icu_mask()`	A.9.2	65	Masks a specified interrupt.
	`icu_unmask()`	A.9.3	66	Unmasks a specified interrupt.
Multi-Function Timer (MFT)	`mft_init()`	A.10.1	67	Initializes the MFT hard-ware to generate a clock tick every specified period of time.
	`mft_delay()`	A.10.2	68	Loads an MFT counter with a specified count, waits for the counter to underflow, then returns.
Microwire Interface	`mw_init`	A.11.1	69	Initializes the Microwire hardware and the Microwire driver.
	`mw_tx`	A.11.2	70	Transmits a specified number (or less) of bytes.
	`mw_rx`	A.11.3	70	Reads a specified number (or less) of received bytes
	`mw_tx_rx`	A.11.4	71	Transmits a specified number of bytes and receives the same amount of bytes at the same time.
System Configu-ration	`SYSCFGEnableIoExpansion()`	A.12.1	72	Enables the BIU IO expansion zone that is used for external I/O.

Table A-1. On-Chip Peripheral Driver Functions (Continued)

Module	Function Call	Section	Page	Description
System Timer	`timer_init()`	A.13.1	73	Initializes the system timer hardware and driver. Currently configured for using the TWM, but other timers could be used.
	`timer_set_wakeup()`	A.13.2	74	Sets up a wakeup call.
	`timer_clear_wakeup()`	A.13.3	75	Clears a wakeup call setting.
	`timer_wakeup_check()`	A.13.4	75	Scans the wakeup list, updates the time counts, and calls the wakeup function for any entry that expires.
	`timer_wait()`	A.13.5	76	Delays execution for a specified number of ticks.
Triple Clock and Reset Module	`TCRSetClock()`	A.14.1	77	Sets the frequency of the system's main clock.
	`TCRPllEnable()`	A.14.2	77	Enables the on-chip PLL.
	`TCRPllDisable()`	A.14.3	78	Disables the on-chip PLL.
Timing and Watchdog Module (TWM)	`twm_init()`	A.15.1	79	Initializes the TWM peripheral and the TWM driver.
	`wd_init()`	A.15.2	79	Initializes the Watchdog.
	`wd_restart()`	A.15.3	80	Restarts the Watchdog.

Table A-1. On-Chip Peripheral Driver Functions (Continued)

Module	Function Call	Section	Page	Description
UART Interface (Inter-rupt mode)	`usart_init()`	A.16.1	81	Initializes the UART to the specified bit rate.
	`usart_shutdown()`	A.16.2	82	Shuts down the UART and releases its resources.
	`usart_tx()`	A.16.3	83	Transmits a specified number (or less) of bytes.
	`usart_rx()`	A.16.4	84	Reads a specified number (or less) of received bytes.
	`usart_rx_count()`	A.16.5	85	Returns the number of bytes that can be currently read from the UART driver.
UART Interface (DMA mode)	`usart_dma_init()`	A.17.1	86	Initializes the UART to the specified bit rate.
	`usart_dma_tx()`	A.17.2	87	Transmits a specified number (or less) of bytes.
	`usart_dma_rx()`	A.17.3	88	Reads a specified number (or less) of received bytes.
	`usart_dma_rx_count()`	A.17.4	88	Returns the number of bytes that can be currently read from the UART-DMA driver.

Table A-1. **On-Chip Peripheral Driver Functions (Continued)**

Module	Function Call	Section	Page	Description
Universal Serial Bus (USB)	`usb_init()`	A.18.1	89	Initialize the USB controller, attach to USB bus, perform enumeration and configuration.
	`usb_shutdown()`	A.18.2	90	Detach from bus, disable USB interrupts, halt USB controller.
	`node_resume()`	A.18.3	90	Send resume signal.
	`get_node_status()`	A.18.4	91	Returns the current state of the USB node.
	`handle_std_usb_request()`	A.18.5	91	Selects and calls routines for handling standard USB requests.
Versatile Timer Unit (VTU)	`vtu_init()`	A.19.1	92	Initializes a specified VTU channel.

A.2 ACCESS.bus (Master Mode)

A.2.1 acbm_init()

Description:	This function initializes the ACCESS.bus hardware and the ACCESS.bus driver.	
Function:	`acbm_init()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`acb_num:`	The unit number of the ACCESS.bus block to be initialized. If the CP3000 device has only one ACCESS.bus block, its unit number is 0.
`unsigned short`	`divisor`	The ratio between the system clock frequency and the desired ACCESS.bus clock frequency. For accuracy this number should be a multiple of 4. It also must be in the range of 32 to 2044.
`void`	`(*callback_p)(void)`	A pointer to a callback function that will be called once a transmit or receive transaction is completed or aborted.

A.2.2 acbm_tx()

Description:	This function transmits data to a specified slave device.	
Function:	`acbm_tx()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`acb_num:`	The unit number of the ACCESS.bus block to be used. If the CP3000 device has only one ACCESS.bus block, its unit number is 0.
`unsigned char`	`address`	The ACCESS.bus address of the slave device to be addressed.
`unsigned char`	`*data`	A pointer to a memory buffer where the data to be transmitted is stored.
`unsigned short`	`size`	The size of the data to be transmitted in bytes.
`boolean`	`stop`	A boolean flag indicating whether a STOP condition should be generated at the end of data transmission.

A.2.3 acbm_rx()

Description:	This function receives data from a specified slave device.	
Function:	`acbm_rx()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`acb_num:`	The unit number of the ACCESS.bus block to be used. If the CP3000 device has only one ACCESS.bus block, its unit number is 0.
`unsigned char`	`address`	The ACCESS.bus address of the slave device to be addressed.
`unsigned char`	`*inbuf`	A pointer to a memory buffer where the data to be transmitted is stored.
`unsigned short`	`size`	The size of the data to be transmitted in bytes.
`boolean`	`stop`	A boolean flag indicating whether a STOP condition should be generated at the end of data transmission.

A.3 ACCESS.bus (Slave Mode)

A.3.1 acbs_init()

Description:	This function initializes the ACCESS.bus hardware and the ACCESS.bus driver.	
Function:	`acbs_init()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned char`	`address`	The 7-bit ACCESS.bus address of the local device.

A.3.2 acbs_tx()

Description:	This function deposits data to be transmitted by the slave the next time the master requests the slave to transmit.	
Function:	`acbs_tx()`	
Return:	`unsigned int`	The number of bytes that were actually transmitted. Could be less than the requested number.
Parameters:		
Type	**Name**	**Description:**
`unsigned char`	`*data`	A pointer to a memory buffer where the data to be transmitted is stored.
`unsigned short`	`size`	The size of the data to be deposited in bytes. Note: when the master asks the slave to transmit, it is up to the master to decide how many bytes it wants to pull. The application is responsible for depositing enough data so that the master will not starve. The amount of data needed by the master is application dependent.

A.3.3 acbs_rx()

Description:	This function reads a specified number (or less) of received bytes into a specified memory location.	
Function:	`acbs_rx()`	
Return:	`unsigned int`	The number of bytes that were actually read. Could be less than the requested number.
Parameters:		
Type	**Name**	**Description:**
`unsigned char`	`*inbuf`	A pointer to the start address of the buffer to receive the data.
`unsigned short`	`size`	The number of bytes to be read.

A.3.4 acbs_rx_count()

Description:	This function returns the number of bytes that can be currently read from the ACCESS.bus slave driver.	
Function:	`acbs_rx_count()`	
Return:	`unsigned int`	The number of bytes that can be currently read from the UART driver.
Parameters:	None.	

A.4 Analog/Digital Converter (ADC)

A.4.1 AdcInit()

Description:	This function initializes the ADC peripheral and the ADC driver.	
Function:	`AdcInit()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`boolean`	`Diff`	Specifies operation in differential or single-ended mode. TRUE is differential mode, and FALSE is single-ended mode.
`unsigned char`	`Channel`	Specifies input channel in single-ended mode, or input channel pair in differential mode.
`unsigned short`	`PRef`	Specifies positive voltage reference. Valid values: PREF_CFG_INTERNAL PREF_CFG_VREFP PREF_CFG_ADC0 PREF_CFG_ADC1
`unsigned short`	`NRef`	Specifies negative voltage reference. Valid values: NREF_CFG_INTERNAL NREF_CFG_ADC2 NREF_CFG_ADC3
`boolean`	`ExternalWakeup`	Specifies whether the ADC conversion is initiated by an external trigger. TRUE is an external trigger, FALSE is a software trigger.

`boolean`	**`TouchScreen`**	Specifies if the ADC is being used in a touchscreen application. In a touchscreen application, the ADC block is configured to be in Pen Down Detection mode at the end of the initialization.
`boolean`	**`Repeat`**	When an external trigger is used, specifies whether the trigger is one-shot or retriggerable. TRUE starts a conversion on every ASYNC trigger. FALSE only starts a conversion on the next ASYNC trigger.
`void (*adc_result_callback_p)(short *Result)`		A pointer to a function that will be called every time a result is available. If the pointer is not used (set to NULL), then the ADC interrupt will not be generated at the end of a conversion. The application can read the conversion result by calling the blocking function **`AdcReadResult()`**.

A.4.2 AdcShutDown()

Description:	This function shuts down the ADC block and disables the ADC interrupt.
Function:	`AdcShutDown()`
Return:	`void`
Parameters:	None.

A.4.3 AdcTouchScreenConfig()

Description:	This function sets up the ADC to sample the appropriate coordinate for the touchscreen application. After a set of coordinates have been sampled, it is the responsibility of the application to configure the ADC for Pen Down Detection mode.	
Function:	`AdcTouchScreenConfig()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned short`	`CoOrdinate`	Specifies which coordinate to sample next. Valid values: SAMPLE_X SAMPLE_Y SAMPLE_Z PEN_DOWN_DETECT SAMPLE_Z_PRE_PENDOWN.

A.4.4 AdcStartConversion()

Description:	This function starts an ADC conversion or primes the ADC to start a conversion on the next rigger event. If the external trigger is not enabled, calling this function starts a conversion. If the external trigger is enabled, this function primes the ADC to start a conversion on the next ASYNC trigger.	
Function:	`AdcStartConversion()`	
Return:	`void`	
Parameters:	None.	

A.4.5 AdcReadResult()

Description:	This function reads the result after the ADC conversion has finished. If during the ADC initialization, there was no callback function provided to read the results from the ADC, then this function can be used to read the results in a busy waiting loop.	
Function:	`AdcReadResult()`	
Return:	`short`	12-bit ADC result.
Parameters:	None.	

A.5 Advanced Audio Interface (AAI)

A.5.1 audio_init()

Description:	This function configures a given memory or I/O zone according to specified parameters.	
Function:	`audio_init()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`audio_config_t`	`*config`	Pointer to an audio configuration structure.
`dl_t`	`*transmit_dl`	A pointer to data link (circular buffer) control structure to be used for voice transmission (audio output).
`dl_t`	`*receive_dl`	A pointer to data link (circular buffer) control structure to be used for voice reception (audio input).
`void`	`(*callback_p)(unsigned int)`	A pointer to a callback function to be called whenever input voice samples are written to the receive data link circular buffer.

A.5.2 audio_set_control_bits()

Description:	This function specifies the number and contents of the control bits that are appended to each transmitted word. Typically, this is used by codecs for volume control.	
Function:	`audio_set_control_bits()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`n_bits`	The number of bits to be appended (0-3).
`unsigned char`	`value`	The value or contents of the control bits (0x0-0x7).

A.5.3 audio_start()

Description:	This function starts the AAI and CVSD/PCM hardware.
Function:	`audio_start()`
Return:	`void`
Parameters:	None.

A.5.4 audio_stop()

Description:	This function stops the AAI and CVSD/PCM hardware.
Function:	`audio_stop()`
Return:	`void`
Parameters:	None.

A.6 Bus Interface Unit (BIU)

A.6.1 biu_config_zone()

Description:	This function configures a given memory or I/O zone according to specified parameters.	
Function:	`biu_config_zone()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`zone`	The zone id. One of **`ZONE0`**, **`ZONE1`**, **`ZONE2`**, or **`IOZONE`**.
`unsigned int`	`ws`	Number of wait states to be applied to this zone.
`unsigned int`	`hs`	Number of hold states to be applied to this zone.
`boolean`	`fr`	A fast-read flag. If TRUE fast-read mode will be enabled for the zone. Ignored for the I/O zone.

A.6.2 biu_set_early_write()

Description:	This function sets the early-write mode of the BIU.	
Function:	`biu_set_early_write()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`boolean`	`er`	Boolean flag to indicate the early-write mode: TRUE - early-write. FALSE - late-write.

A.7 Controller Area Network (CAN)

A.7.1 can_init()

Description:	Initializes the CAN hardware and CAN driver. After returning from this function, the individual CAN message buffers are ready to be configured for transmission and reception. Puts message buffers into benign state and zeros all contents.	
Function:	`can_init()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`can_id`	Unit number of the CAN interface. For CP3000 devices with only one CAN unit, the unit number is 0.
`can_config_t`	`global_config`	Sets message buffers into locking mode, and configures direction of data byte. Values ORed together.
`unsigned int`	`bus_freq_ratio`	Sets CAN bus bit rate relative to CPU frequency. Value is ratio (system freq)/(CAN bus freq). For example, for a 12 MHz CPU and 50 KHz CAN bus bit rate, bus_freq_ratio = 240.
`unsigned short`	`global_mask`	Sets standard message mask for message buffers 0-14, 1 indicates don't care for similar bit in ID1. RTR bit is automatically masked when a remote message is sent to ensure reception of response.
`unsigned short`	`global_mask_ext`	Sets standard message mask for message buffers 0-14, 1 indicates don't care for similar bit in ID0. RTR bit is automatically masked when a remote message is sent to ensure reception of response.
`unsigned short`	`basic_mask`	Sets standard message mask for message buffer 15, 1 indicates don't care for similar bit in ID1.

unsigned short	**basic_mask_ext**	Sets standard message mask for message buffer 15, 1 indicates don't care for similar bit in ID0.
void	**(*mbuf_callback) (unsigned short mbuf_num, unsigned short cediag)**	Callback function invoked whenever interrupt on any message buffer is activated. Setting this to a non-null value causes setup of interrupts for this CAN block. A null value disables interrupts. If the **cediag** parameter is 0, then the **mbuf_num** parameter indicates which message buffer caused the interrupt (which could be due to a receive or transmit event). If the **cediag** parameter is non-zero, **cediag** is a copy of the CEDIAG register, which provides detailed information about the error.

A.7.2 can_config_tx()

Description:	Configure a message buffer to transmit a packet, any previous configuration is completely overridden. There are five modes (see **`tx_type`** parameter and **`can_tx_type_t`**).	
Function:	`can_config_tx()`	
Return:	`unsigned int`	ERR_MBUF_BUSY if the buffer is busy already. To force a new setup, call with TX_CANCEL first.

Parameters:

Type	**Name**	**Description:**
`unsigned int`	`can_id`	Unit number of the CAN interface. For CP3000 devices with only one CAN unit, the unit number is 0.
`unsigned char`	`mbuf_num`	Selects message buffer to set into this mode, 0-14.
`unsigned long`	`id`	The Message Identifier.
`unsigned char`	`*msg`	Pointer to an array of up to 8 bytes of payload.
`unsigned int`	`len`	Length of payload.
`msg_type_t`	`msg_type`	Normal or extended.
`unsigned int`	`priority`	The priority to set the message to.
`can_tx_type_t`	`tx_type`	SEND_DATA: Send a single data frame. REMOTE_REQ: Send a remote frame and setup to receive the response, user calls **`can_read_msg()`** to retrieve the response from the message buffer. RESPOND_TO_REMOTE: Loads the message buffer with data, etc. and waits for a matching Remote frame before transmitting. SEND_THEN_RTR: Same as RESPOND_TO_REMOTE but send the frame once before waiting for an incoming Remote frame. TX_CANCEL: Cancel any pending transmit or RESPOND_TO_REMOTE setting.

A.7.3 can_config_rx()

Description:	Configure the message buffer to receive a Data Frame only. Call **`can_read_msg()`** to retrieve the payload data and details of message actually received.	
Function:	`can_config_rx()`	
Return:	`unsigned int`	ERR_MBUF_BUSY if the buffer is already busy, otherwise zero.
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`can_id`	Unit number of the CAN interface. For CP3000 devices with only one CAN unit, the unit number is 0.
`unsigned char`	`mbuf_num`	Selects message buffer to set into this mode, 0-14.
`unsigned long`	`id`	The Message Identifier.
`msg_type_t`	`msg_type`	Normal or extended.

A.7.4 can_read_msg()

Description:	Reads a message from the specified message buffer (0-14).	
Function:	`can_read_msg()`	
Return:	`unsigned int`	Returns length of payload.
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`can_id`	Unit number of the CAN interface. For CP3000 devices with only one CAN unit, the unit number is 0.
`unsigned char`	`mbuf_num`	Selects message buffer to read, 0-14.
`unsigned long`	`*id`	The Message Identifier is returned in this parameter.
`unsigned char`	`*msg`	Pointer to array of up to 8 bytes to write payload into priority.
`msg_type_t`	`*msg_type`	Normal or extended is returned in this pointer parameter.
`unsigned int`	`*priority`	Pointer to the priority of the message received.
`unsigned short`	`*tstamp`	Bit time stamp of message reception is returned in this pointer parameter.

A.7.5 can_shutdown()

Description:	Shuts down the CAN driver and specified CAN interface. Also disables CAN interrupts. To use the specified CAN interface subsequently, call `can_init()`.	
Function:	`can_shutdown()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`can_id`	Unit number of the CAN interface. For CP3000 devices with only one CAN unit, the unit number is 0.

A.8 Flash Memory Interface

A.8.1 FlashInit()

Description:	This function initializes the flash memory timing registers for the specified frequency.	
Function:	`FlashInit()`	
Return:	`void`	
Parameters:		
`memory_t`	`Type`	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA
`frequency_t`	`Frequency`	Specifies the frequency. Valid values: FREQ_8MHz FREQ_12MHz FREQ_16MHz FREQ_20MHz FREQ_24Mhz

A.8.2 FlashGetPageSize()

Description:	This function returns the page size of the specified flash memory.	
Function:	`FlashGetPageSize()`	
Return:	`unsigned short`	The page size.
Parameters:		
`memory_t`	`Type`	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA

A.8.3 FlashGetPageNum()

Description:	This function returns the number of flash pages of the specified flash memory.	
Function:	`FlashGetPageNum()`	
Return:	`unsigned short`	The number of pages.
Parameters:		
`memory_t`	`Type`	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA

A.8.4 FlashGetStartAddress()

Description:	This function returns the start address of the specified flash memory.	
Function:	**`FlashGetStartAddress()`**	
Return:	**`unsigned long`**	The start address.
Parameters:		
`memory_t`	**`Type`**	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA

A.8.5 FlashInformationBlockRead()

Description:	This function reads a specified number of words from a location in Information Block 0, 1, or 2. Information Block data is read through the register-based interface. Only word read operations are supported, and the Offset must be word-aligned (LSB = 0). The Type controls whether Information Block 0 or 1 (FLASH_PROGRAM) or Information Block 2 (FLASH_DATA) is accessed. The Offset controls whether Information Block 0 (000h–07Eh) or Block 1 (080h–0FEh) is accessed. The Offset used to access Information Block 2 is 000h–07Eh.	
Function:	`FlashInformationBlockRead()`	
Return:	`unsigned short`	The number of words read.
Parameters:		
`memory_t`	`Type`	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA
`unsigned short`	`Offset`	The byte offset to the first source word.
`unsigned short`	`*Data`	A pointer to a buffer of words to store the data read from the Information Block.
`unsigned short`	`Size`	Number of words to read.

A.8.6 FlashInformationBlockWrite()

Description:	This function writes a specified number of words to a location in Information Block 1 or 2. Writing is only allowed when global write protection is disabled and the write enable bit is set for the sector which contains the word to be written. Information Block 0 cannot be written by the CPU. Information Block data is written through the register-based interface. Only word write operations are supported, and the Offset must be word-aligned (LSB = 0). The Type controls whether Information Block 1 (FLASH_PROGRAM) or Information Block 2 (FLASH_DATA) is accessed. The Offset is 080h–0FEh for Information Block 1 or 000h–07Eh for Information Block 2.	
Function:	`FlashInformationBlockWrite()`	
Return:	`unsigned short`	The number of words written.
Parameters:		
`memory_t`	`Type`	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA
`unsigned short`	`Offset`	The byte offset to the first destination word.
`unsigned short`	`*Data`	A pointer to a buffer of words holding the data to be written to the Information Block.
`unsigned short`	`Size`	Number of words to write.

A.8.7 FlashInformationBlockErase()

Description:	Erases an Information Block and the corresponding Main Block. Page erase is not supported for Information Blocks.	
Function:	`FlashInformationBlockErase()`	
Return:	`void`	
Parameters:		
`memory_t`	`Type`	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA
`unsigned short`	`Offset`	Any Offset within the Information Block.
`unsigned char`	`Data`	Any data (don't care).

A.8.8 FlashRead()

Description:	This function reads a specified number of byte from either flash memory. The Type controls whether flash program or flash data memory is accessed. The Offset is added to the flash memory base address.	
Function:	`FlashRead()`	
Return:	`void`	
Parameters:		
`memory_t`	`Type`	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA
`unsigned long`	`Offset`	The byte offset to the first source byte.
`unsigned short`	`Size`	Number of bytes to read.
`unsigned char`	`*Data`	A pointer to a buffer of bytes to store the data read from the flash memory.

A.8.9 FlashPageProgramSafe()

Description:	This function provides a generalized mechanism to write to the flash memory Main Blocks. It assumes that the data to be written is not necessarly aligned on a page boundary. The write cycle implies that an erase operation has taken place prior to programming. The following algorithm is used: a) Locate a place to save a page of data. c) Erase it to allow a later writting operation. b) Copy the current page to save area. d) Erase current page. e) Write current page with data from saved and new data (which allows partial page writes). Temporary storage is allocated from the heap. If the request fails, the last page of the Main Block is used.	
Function:	`FlashPageProgramSafe()`	
Return:	`void`	
Parameters:		
`memory_t`	`Type`	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA
`unsigned long`	`Offset`	The byte offset to the first destination byte.
`unsigned short`	`Size`	Number of bytes to program.
`unsigned char`	`*InBuf`	Buffer containing data to program to flash.

A.8.10 FlashPageProgram()

Description:	This function always writes one page of data. The data is obtained from an input buffer (InBuf) and a reference buffer (RefBuf). The Size specifies the number of bytes from a page offset up to the end of the page and therefore must not exceed the page size. The page offset is calculated from the specified offset with the specified Main Block(s). The algorithm used here is complicated by the fact that the size of the modified data is not always exactly a page. Because the whole page must be written, the page must be saved to the reference buffer, the page is erased, and then the page is written with the saved and modified data. The page must be saved and erased before this function is called. Writing is only allowed when global write protection is disabled. Writing by the CPU is only allowed when the write enable bit is set for the sector which contains the word to be written. The CPU cannot write the Boot Area. Only word-wide write access to word-aligned addresses is supported.	
Function:	`FlashPageProgram()`	
Return:	`void`	
Parameters:		
`memory_t`	`Type`	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA
`unsigned long`	`Offset`	The byte offset to the first modified destination byte.
`unsigned short`	`Size`	Number of bytes to modify.
`unsigned char`	`*RefBuf`	A reference buffer where the old contents of the page to write have been saved. Possibly null, if the whole page is being written.
`unsigned char`	`*InBuf`	A buffer containing the new data to be written.

A.8.11 FlashEraseAll()

Description:	This function clears all of the specified flash memory, which sets every byte to FFh.	
Function:	`FlashEraseAll()`	
Return:	`void`	
Parameters:		
`memory_t`	`Type`	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA

A.8.12 FlashPageErase()

Description:	A flash erase operation sets all of the bits in the erased region. Pages of a main block can be individually erased if their write enable bits are set. Each page in Main Blocks 0 and 1 consists of 1024 bytes (512 words). Each page in Main Block 2 consists of 512 bytes (256 words).	
Function:	`FlashPageErase()`	
Return:	`void`	
Parameters:		
`memory_t`	`Type`	Specifies the flash memory. Valid values: FLASH_PROGRAM FLASH_DATA
`unsigned long`	`Address`	Address of the page to erase

A.9 Interrupt Control Unit (ICU)

A.9.1 icu_init()

Description:	This function initializes the ICU hardware. It masks all interrupts.
Function:	`icu_init()`
Return:	`void`
Parameters:	None.

A.9.2 icu_mask()

Description:	This function masks a single interrupt specified by its IRQ number.	
Function:	`icu_mask()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`irq`	The IRQ number of the interrupt to be masked.

A.9.3 icu_unmask()

Description:	This function unmasks a single interrupt specified by its IRQ number.	
Function:	`icu_unmask()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`irq`	The IRQ number of the interrupt to be unmasked.

A.10 Multi-Function Timer (MFT)

A.10.1 mft_init()

Description:	This function initializes the MFT hardware to generate a clock tick every specified period of time.	
Function:	`mft_init()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`const int`	`mode`	Selects the mode of operation.
`const int`	`ClockTick1`	Mode 1: The high/low pulse width. Mode 3: The required length of period between clock ticks in milliseconds for timer I.
`const int`	`ClockTick2`	Modes 1 and 3: The required length of period between clock ticks in milliseconds for timer II.
`const int`	`ClockTick3`	Mode 1: The high/low pulse width. Mode 3: Unused.

A.10.2 mft_delay()

Description:	This function loads an MFT counter with the specified delay count, waits for the counter to underflow, then returns.	
Function:	`mft_delay()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`const int`	`mode`	Selects the mode of operation.
`const int`	`Channel`	Selects channel 1 or 2.
`unsigned short`	`Dly`	Specifies delay count.

A.11 Microwire Interface

A.11.1 mw_init()

Description:	This function initializes the Microwire hardware and the Microwire driver. After returning from this function the Microwire interface is ready to transmit and receive data.	
Function:	`mw_init()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`boolean`	`master_mode`	A boolean value that determines whether the Microwire interface should be configured to master mode or slave mode.
`unsigned int`	`divisor`	A number that specifies the clock rate used by the Microwire interface when in master mode. This number actually specifies the ratio between the system main clock and the clock that will be used by the Microwire interface. The larger the number, the slower the Microwire clock. Only even numbers in the range 2-256 are acceptable. Any other value will be truncated. This argument is ignored in slave mode.
`void`	`(*callback_p)(void)`	A pointer to a void function with no arguments that will be called when a transmit request is completed, receive request is completed, or transmit/receive request is completed.

A.11.2 mw_tx()

Description:	This function transmits a specified number (or less) of bytes starting from a specified memory location.	
Function:	`mw_tx()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned char`	`*buf`	A pointer to a buffer that contains the data to be transmitted.
`unsigned int`	`size`	The number of bytes to be transmitted.

A.11.3 mw_rx()

Description:	This function reads a specified number (or less) of received bytes into a specified memory location.	
Function:	`mw_rx()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned char`	`*buf`	A pointer to the start address of the buffer to receive the data.
`unsigned int`	`size`	The number of bytes to be read.

A.11.4 mw_tx_rx()

Description:	This function transmits a specified number of bytes and receives the same amount of bytes at the same time.	
Function:	`mw_tx_rx()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned char`	`*rxbuf`	A pointer to the start address of the buffer to receive the data.
`unsigned char`	`*txbuf`	A pointer to a buffer that contains the data to be transmitted.
`unsigned int`	`size`	The number of bytes to be transmitted and received.

A.12 System Configuration

A.12.1 SYSCFGEnableIoExpansion()

Description:	This function enables the BIU IO expansion zone that is used for external I/O. Note that in IRE mode, I/O expansion is disabled by default, and therefore it is necessary to call this function.
Function:	`SYSCFGEnableIoExpansion()`
Return:	`void`
Parameters:	None.

A.13 System Timer

A.13.1 timer_init()

Description:	This function initializes the TWM hardware and the timer driver, specifically the wakeup list.
Function:	`timer_init()`
Return:	`void`
Parameters:	None.

A.13.2 timer_set_wakeup()

Description:	This function sets up a wakeup call. It does this by inserting an entry to the wakeup list.	
Function:	`timer_set_wakeup()`	
Return:	`int`	A handle to be used for this wakeup call for further references (e.g., if the wakeup call is to be cancelled later). In practice, this is the index of the wakeup call in the wakeup list.
Parameters:		
Type	**Name**	**Description:**
`void`	`(* func)(void)`	A pointer to a function that will be called when the time expires.
`unsigned int`	`clock_ticks`	The number of clocks ticks to elapse before the time expires.
`boolean`	`repeat`	A flag that specifies whether this is a repetitive wakeup call or a one-time wakeup call. If FALSE, there will be a one-time wakeup call and then the wakeup entry will be invalidated. If TRUE, the time count will restart automatically when the time expires.

A.13.3 timer_clear_wakeup()

Description:	This function deactivate a specified wakeup call.	
Function:	`timer_clear_wakeup()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`int`	`index`	The wakeup call handle (index).

A.13.4 timer_wakeup_check()

Description:	This function scans the wakeup list, increments time counts, and calls the wakeup function for any entry that expires.
Function:	`timer_clear_wakeup()`
Return:	`void`
Parameters:	None.

A.13.5 timer_wait()

Description:	This function provides a wait (delay) service. It delays the execution of the program for a give number of ticks. Note: The implementation is different between the OS and OS-less versions. In the OS version, the current task is delayed for the required number of clock ticks (but other tasks may run in the mean time). In the OS-less version, there is a busy wait which is implemented by polling a flag that will become TRUE after the required number of clock ticks.	
Function:	`timer_wait()`	
Return:	`boolean`	Indicates whether the wait was completed successfully.
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`ticks`	The number of clock ticks to wait/delay.

A.14 Triple Clock and Reset

A.14.1 TCRSetClock()

Description:	This function sets the frequency of the system's main clock.	
Function:	`TCRSetClock()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`freq_t`	`freq`	The required frequency. This is a value that has several possible values for different clock frequencies.

A.14.2 TCRPllEnable()

Description:	This function enables the on-chip PLL. It also selects the PLL output to be used as input for the high frequency clock generator (should this be separated from PLL).
Function:	`TCRPllEnable()`
Return:	`void`
Parameters:	None.

A.14.3 TCRPllDisable()

Description:	This function disables the on-chip PLL.
Function:	`TCRPllDisable()`
Return:	`void`
Parameters:	None.

A.15 Timer Watchdog Module (TWM)

A.15.1 twm_init()

Description:	This function initializes the TWM peripheral and the TWM driver.	
Function:	`twm_init()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`tick_period`	The length of the clock tick generated by the timing module in milliseconds.
`void`	`(*tick_callback_p)(void)`	A pointer to a function that will be called on every clock tick.

A.15.2 wd_init()

Description:	This function initializes Watchdog.	
Function:	`wd_init()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned char`	`ticks`	This is the number of clock ticks the Watchdog counter will be loaded with. The Watchdog will count down and unless wd_restart is called early enough, it will reach zero and reset the CPU.

A.15.3 wd_restart()

Description:	This function restarts the Watchdog. This causes the Watchdog counter to be reloaded with the original value that was assigned to it and start the countdown again. If this is not done early enough, the Watchdog Error signal will be triggered, and that will reset the CPU.
Function:	`wd_restart()`
Return:	`void`
Parameters:	None.

A.16 UART Interface (Interrupt Mode)

A.16.1 usart_init()

Description:	This function initializes the UART hardware and the UART driver. After this function returns, the UART is ready to transmit and receive data.
Function:	`usart_init()`
Return:	`void`

Parameters:

Type	**Name**	**Description:**
`unsigned int`	`usart_num`	The unit number of the UART to be initialized. If the CP3000 device has only one UART, the unit number is 0.
`bitrate_t`	`bitrate`	The required communication bit rate. Check the bitrate_t type for supported bit rates.
`flow_control_t`	`flow_control`	Indicates what type of flow control is to be used. Check the flow_control_t type for flow control types. Note: software flow control is not yet supported.
`void`	`(*rx_callback)(void)`	An optional pointer. This function will automatically be called when data is received unless the pointer is NULL. This function should be as short as possible as it is called from the UART receive interrupt handler.

A.16.2 usart_shutdown()

Description:	This function shuts down the specified UART and releases its resources.	
Function:	`usart_shutdown()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`usart_num`	The unit number of the UART to be shut down. If the CP3000 device has only one UART, the unit number is 0.

A.16.3 usart_tx()

Description:	This function transmits a specified number (or less) of bytes starting from a specified memory location.	
Function:	`usart_tx()`	
Return:	`unsigned int`	The number of bytes that were actually transmitted. Could be less than the requested number.
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`usart_num`	The unit number of the UART to be used. If the CP3000 device has only one UART, the unit number is 0.
`unsigned char`	`*data`	A pointer to the start address of the data to be transmitted.
`unsigned int`	`size`	The number of bytes to be transmitted.

A.16.4 usart_rx()

Description:	This function reads a specified number (or less) of received bytes into a specified memory location.	
Function:	`usart_rx()`	
Return:	`unsigned int`	The number of bytes that were actually read. Could be less than the requested number.
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`usart_num`	The unit number of the UART to be used. If the CP3000 device has only one UART, the unit number is 0.
`unsigned char`	`*data`	A pointer to the start address of the buffer to receive the data.
`unsigned int`	`size`	The number of bytes to be read.

A.16.5 usart_rx_count()

Description:	This function returns the number of bytes that can be currently read from the UART driver.	
Function:	`usart_rx_count()`	
Return:	`unsigned int`	The number of bytes that can be currently read from the UART driver.
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`usart_num`	The unit number of the UART to be used. If the CP3000 device has only one UART, the unit number is 0.

A.17 UART Interface (DMA Mode)

A.17.1 usart_dma_init()

Description:	This function initializes the UART according to the required bit rate. It initializes DMA channels 0 for UART receive and channel 1 for UART transmit.	
Function:	`usart_dma_init()`	
Return:		The number of bytes that were actually transmitted. Could be less than the requested number.
Parameters:		
Type	**Name**	**Description:**
`bitrate_t`	`bitrate`	The required communication bit rate. Check the bitrate_t type for supported bit rates.
`flow_control_t`	`flow_control`	Indicates what type of flow control is to be used. Check the flow_control_t type for flow control types. Note: software flow control is not yet supported.
`void`	`(*rx_callback)(void)`	An optional pointer. This function will automatically be called when data is received unless the pointer is NULL. This function should be as short as possible as it is called from an interrupt handler.

A.17.2 usart_dma_tx()

Description:	This function transmits a specified number (or less) of bytes starting from a specified memory location.	
Function:	`usart_dma_tx()`	
Return:	`unsigned int`	The number of bytes that were actually transmitted. Could be less than the requested number.
Parameters:		
Type	**Name**	**Description:**
`unsigned char`	`*data`	A pointer to the start address of the data to be transmitted.
`unsigned int`	`size`	The number of bytes to be transmitted.

A.17.3 usart_dma_rx()

Description:	This function reads a specified number (or less) of received bytes into a specified memory location.	
Function:	`usart_dma_rx()`	
Return:	`unsigned int`	The number of bytes that were actually read. Could be less than the requested number.
Parameters:		
Type	**Name**	**Description:**
`unsigned char`	`*data`	A pointer to the start address of the buffer to receive the data.
`unsigned int`	`size`	The number of bytes to be read.

A.17.4 usart_dma_rx_count()

Description:	This function returns the number of bytes that can be currently read from the UART-DMA driver	
Function:	`usart_rx_count()`	
Return:	`unsigned int`	The number of bytes that can be currently read from the UART-DMA driver.
Parameters:	None.	

A.18 Universal Serial Bus (USB)

A.18.1 usb_init()

Description:	This function takes the device automatically through the enumeration and configuration process with the host. Subsequent calls to **`usb_rx_data()`** and **`usb_tx_data()`** perform operations at the transfer level, i.e., data transfers smaller or greater than the device maximum packet size. The descriptors required for the enumeration process are configured by constructing the necessary structures in a header file (**descrpt.h**). The driver is capable of working in a simple blocking (synchronous) mode or in a more flexible non-blocking (asynchronous) mode.	
Function:	`usb_init()`	
Return:	`boolean`	Returns TRUE unless the USB module cannot be detected or the PLL has not been enabled.
Parameters:		
Type	**Name**	**Description:**
`usb_init_info_t`	`*usbinitinfo`	Initialization information from user.

A.18.2 usb_shutdown()

Description:	Detaches node from USB bus and shuts down the USB controller and driver.
Function:	`usb_shutdown()`
Return:	`void`
Parameters:	None.

A.18.3 node_resume()

Description:	Sends resume signal on the USB bus, if in the Suspend state.	
Function:	`node_resume()`	
Return:	`boolean`	Returns TRUE if request was successful, otherwise returns FALSE.
Parameters:		
Type	**Name**	**Description:**
`uint8`	`ucState`	Valid values: SIGNAL_RESUME GO_OPERATIONAL

A.18.4 get_node_status()

Description:	Returns the current USB state of the node.	
Function:	`get_node_status()`	
Return:	`usb_node_stat_t`	The current USB state of the node.
Parameters:	None.	

A.18.5 handle_std_usb_request()

Description:	Selects and calls the relevant routine to handle standard USB requests received.	
Function:	`handle_std_usb_request()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`usb_request_t`	`USBRequest`	Structure for holding standard device request data.

A.19 Versatile Timer Unit (VTU)

A.19.1 vtu_init()

Description:	Initializes a specified VTU channel. This function can also be called to update a VTU configuration (e.g. modulating the PWM values). Mode 0 (low power): all VTU timers are stopped. Mode 1 (dual 8-bit PWM): sets the duty and period registers for two 8-bit PWM timers in a single VTU channel. Mode 2 (single 16-bit PWM): sets the duty and period registers for one 16-bit PWM timer in a single VTU channel.	
Function:	`vtu_init()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`const int`	`Channel`	Selects the VTU channel being initialized.
`const int`	`VTUmode`	Selects the mode of operation.
`const int`	`Count`	Initializes the COUNT register in the VTU channel.
`const int` `const int`	`Duty` `Period`	Mode 1: Duty7:0 specifies the duty cycle value for the 8-bit PWM timer 1. Duty15:8 specifies the duty cycle value for the 8-bit PWM timer 2. Period7:0 specifies the period value for the 8-bit PWM timer 1. Period15:8 specifies the period value for the 8-bit PWM timer 2. Mode 2: Duty15:0 specifies the duty cycle value for the 16-bit PWM timer. Period15:0 specifies the period value for the 16-bit PWM timer.
`const int`	`Prescaler`	Prescaler counter value.

Board-Level Peripheral Drivers B

B.1 Summary of Board-Level Peripheral Driver Functions

Table B-1 lists the board-level (off-chip) peripheral driver functions.

Table B-1. Board-Level Peripheral Driver Functions

Module	Function Call	Section	Page	Description
CFI-Compliant External Flash Memory Interface	`CfiFlashGetNumBlocks()`	B.2.1	95	Returns number of blocks in external flash memory.
	`CfiFlashReadReset()`	B.2.2	95	Places external flash memory in Read mode.
	`CfiFlashAutoSelect()`	B.2.3	96	Reads the electronic signature of the device, the manufacturer code, or the protection level of a block.
	`CfiFlashPageErase()`	B.2.4	97	Erases external flash memory blocks.
	`CfiFlashEraseAll()`	B.2.5	98	Erases entire external flash memory.
	`CfiFlashProgram()`	B.2.6	99	Programs external flash memory.
	`CfiFlashQueryGet()`	B.2.7	100	Reads a section of CFI information from a flash device in query mode.
	`CfiFlashReadQuery()`	B.2.8	101	Puts a flash device into query mode, and reads the specified flash information.
	`CfiFlashBlock()`	B.2.9	102	Build an array using the specified block offsets.

Table B-1. **Board-Level Peripheral Driver Functions (Continued)**

Module	Function Call	Section	Page	Description
Codec	`codec_init()`	B.3.1	103	Initializes the codec driver.
	`codec_set_volume()`	B.3.2	104	Sets the volume level on the codec output.
	`codec_adjust_volume()`	B.3.3	104	Adjusts the volume level on the codec output.
	`codec_start()`	B.3.4	105	Starts the codec.
	`codec_stop()`	B.3.5	105	Stops the codec.
EEPROM	`eep_init()`	B.4.1	106	Initializes communication with the EEPROM over the ACCESS.bus interface.
	`eep_buf_read()`	B.4.2	107	Reads from EEPROM.
	`eep_buf_write()`	B.4.3	108	Writes to the EEPROM.
	`eep_total_size()`	B.4.4	108	Returns EEPROM size.
LEDs	`led_init()`	B.5.1	109	Initializes the LEDs.
	`led_set_all()`	B.5.2	109	Sets the binary LEDs.
	`led_set()`	B.5.3	110	Turns on one LED.
	`led_clear()`	B.5.4	110	Turns off one LED.
	`led_toggle()`	B.5.5	111	Toggles one LED.
DIP Switches	`switch_on()`	B.6.1	112	Returns the boolean value of a specified switch.

B.2 CFI-Compliant Flash Interface

B.2.1 CfiFlashGetNumBlocks()

Description:	This function returns the number of blocks in a CFI-compliant external flash memory, as indicated in the global variable NUM_BLOCKS.	
Function:	`CfiFlashGetNumBlocks()`	
Return:	`int`	Number of flash blocks
Parameters:	None.	

B.2.2 CfiFlashReadReset()

Description:	This function places the flash in the Read mode described in the data sheet. In this mode, the flash can be read as normal memory. All of the other functions leave the flash in the Read mode, so this is not strictly necessary. It is provided for completeness. A wait of 10 us is required if the command is called during a program or erase instruction. This is included here to guarantee correct operation. The functions in this library call this function if they suspect an error during programming or erasing, so that the 10 us pause is included. Otherwise, they use the single instruction technique for increased speed.
Function:	`CfiFlashReadReset()`
Return:	`void`
Parameters:	None.

B.2.3 CfiFlashAutoSelect()

Description:	This function can be used to read the electronic signature of the device, the manufacturer code, or the protection level of a block.	
Function:	`CfiFlashAutoSelect()`	
Return:	`int`	When iFunc is >= 0, the function returns CFI_FLASH_BLOCK_PROTECTED if the block is protected (0001h) or CFI_FLASH_BLOCK_UNPROTECTED if it is unprotected (0000h). See the Auto Select command in the data sheet for further information. When iFunc is CFI_FLASH_READ_MANUFACTURER, the function returns the manufacturer's code. The manufacturer code for ST is 0020h. When iFunc is CFI_FLASH_READ_DEVICE_CODE the function returns the Device Code. The device codes for the parts are: M29F160BT 22CCh M29F160BB 224Bh M29W160BT 22C4h M29W160BB 2249h M29W160DT 22C4h M29W160DB 2249h When iFunc is invalid, the function returns FLASH_BLOCK_INVALID (-5).
Parameters:		
Type	**Name**	**Description:**
`int`	`iFunc`	Must be set either to the Read Signature values or to the block number. The header file defines the values for reading the Signature.

B.2.4 CfiFlashPageErase()

Description:	This function erases up to NumBlocks in the flash. The blocks can be listed in any order. The function does not return until the blocks are erased. If any blocks are protected or invalid, none of the blocks are erased. During the Erase Cycle the Data Toggle Flow Chart of the Data Sheet is followed. The polling bit DQ7 is not used.	
Function:	`CfiFlashPageErase()`	
Return:	`int`	The function returns the following conditions: CFI_FLASH_SUCCESS CFI_FLASH_TOO_MANY_BLOCKS CFI_FLASH_MPU_TOO_SLOW CFI_FLASH_WRONG_TYPE CFI_FLASH_ERASE_FAIL The user's array Block[] is used to report errors on the specified blocks. If a time-out occurs because the MPU is too slow, then the blocks in Block[] which are not erased are overwritten with CFI_FLASH_BLOCK_NOT_ERASED (FFh) and the function returns CFI_FLASH_MPU_TOO_SLOW. If an error occurs during the erasing of the blocks, the function returns CFI_FLASH_ERASE_FAIL. If both errors occur, then the function will set the Block[] array to CFI_FLASH_BLOCK_NOT_ERASED for the unerased blocks. It will return CFI_FLASH_ERASE_FAIL even though the CFI_FLASH_MPU_TOO_SLOW has also occurred.
Parameters:		
Type	**Name**	**Description:**
`unsigned char`	`NumBlocks`	The number of blocks in the array Block[].
`unsigned char`	`Block[]`	Array containing the blocks to be erased.

B.2.5 CfiFlashEraseAll()

Description:	This function can be used to erase the whole flash chip, if no blocks are protected. If any blocks are protected, then nothing is erased.	
Function:	`CfiFlashEraseAll()`	
Return:	`int`	On success the function returns CFI_FLASH_SUCCESS. If a block is protected, then the function returns the number of the block and no blocks are erased. If the erase algorithms fails then the function returns CFI_FLASH_ERASE_FAIL. If the wrong type of flash is detected then CFI_FLASH_WRONG_TYPE is returned.
Parameters:		
Type	**Name**	**Description:**
`unsigned char`	`*ucResults`	ucResults is a pointer to an array where the results will be stored. If ucResults == NULL, then no results are stored. Otherwise, the results are written to the array if an error occurs. The array is left unchanged if the function returns CFI_FLASH_SUCCESS. The errors written to the array are: CFI_FLASH_BLOCK_ERASED if the block erased correctly, or CFI_FLASH_BLOCK_ERASE_FAILURE if the block failed to erase.

B.2.6 CfiFlashProgram()

Description:	This function is used to program an array into the flash. It does not erase the flash first and may fail if the block(s) are not erased first.	
Function:	`CfiFlashProgram()`	
Return:	`int`	The function returns the following conditions: CFI_FLASH_SUCCESS CFI_FLASH_PROGRAM_FAIL CFI_FLASH_OFFSET_OUT_OF_RANGE CFI_FLASH_WRONG_TYPE On success the function returns CFI_FLASH_SUCCESS. If a programming failure occurs, the function returns CFI_FLASH_PROGRAM_FAIL. If the address range to be programmed exceeds the address range of the flash device, the function returns CFI_FLASH_OFFSET_OUT_OF_RANGE and nothing is programmed. If the wrong type of flash is detected, then CFI_FLASH_WRONG_TYPE is returned and nothing is programmed. If part of the address range to be programmed falls within a protected block, the function returns the number of the first protected block encountered and nothing is programmed.
Parameters:		
Type	**Name**	**Description:**
`unsigned long`	`ulOff`	Word offset into the flash memory to be programmed.
`size_t`	`NumWords`	The number of words in the array.
`void`	`*Array`	Pointer to the array to be programmed.

B.2.7 CfiFlashQueryGet()

Description:	CFI information relative to the flash part are contained in various sections (see data sheet for details). This routine reads a single section once the part is in query mode. The section address buffer pointer and size are provided to read the specified data.	
Function:	`CfiFlashQueryGet()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`uint`	`baseAddr`	Base address of the flash memory device.
`uint16`	`*ptr`	Pointer to buffer to receive flash data.
`uint`	`size`	Number of bytes to be read.

B.2.8 CfiFlashReadQuery()

Description:	Put the flash device into query mode, and read the specified flash information.	
Function:	`CfiFlashReadQuery()`	
Return:	`int`	A return code indicating results.
Parameters:		
Type	**Name**	**Description:**
`query_t`	`query`	A type of query corresponding to the info area (see data sheet).
`uint16`	`*resultBuf`	Pointer to a buffer for receiving the section data.

B.2.9 CfiFlashBlock()

Description:	Build an array using the block offsets.	
Function:	`CfiFlashBlock()`	
Return:	`int`	0 indicates success. -1 indicates a failure for an address specified below the base.
Parameters:		
Type	**Name**	**Description:**
`uint8`	`*address`	Memory address where the array of blocks starts.
`uint32`	`numBlock`	Number of blocks in the resulting array.
`uint8`	`*result`	Pointer to the array.

B.3 Codec

B.3.1 codec_init()

Description:	This function initializes the codec driver.
Function:	`codec_init()`
Return:	`void`

Parameters:

Type	**Name**	**Description:**
`dl_t`	`*xmt_dl`	A pointer to data link (circular buffer) control structure for voice transmit (audio output).
`dl_t`	`*rcv_dl`	A pointer to data link (circular buffer) control structure for voice reception (audio input).
`void`	`(*rx_callback_p)(unsigned int)`	A pointer to a callback function that is called whenever input voice samples are written to the receive data link circular buffer.
`audio_conv_t`	`conv`	Indicates type of conversion for the CVSD/PCM conversion block on both transmit and recive paths. This optional conversion may complement the codec functionality when audio encoding formats other than the codec's have to be used by the application. Valid values are: NOCVSDCONV CVSD2ULAW CVSD2ALAW CVSD2LINEARPCM

B.3.2 codec_set_volume()

Description:	This function sets the volume level on the codec output.	
Function:	`codec_set_volume()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`volume`	A value in the range of 0-7, in which 0 is low volume and 7 is high volume.

B.3.3 codec_adjust_volume()

Description:	This function adjusts the volume level on the codec output.	
Function:	`codec_adjust_volume()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`int`	`delta`	A value to be added to the current volume level. This number can be positive (to increase volume) or negative (to decrease volume). The volume level will remain in the range of 0-7.

B.3.4 codec_start()

Description:	This function starts the codec.
Function:	`codec_start()`
Return:	`void`
Parameters:	None.

B.3.5 codec_stop()

Description:	This function stops the codec.
Function:	`codec_stop()`
Return:	`void`
Parameters:	None.

B.4 EEPROM

B.4.1 eep_init()

Description:	This function initializes the communication with the EEPROM over the ACCESS.bus interface.
Function:	`eep_init()`
Return:	`void`
Parameters:	None.

B.4.2 eep_buf_read()

Description:	This function reads a specified number of bytes from the EEPROM.	
Function:	`eep_buf_read()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned short`	`address`	The address (offset) inside the EEPROM from which data should be read.
`unsigned char`	`*buf`	A pointer to a data buffer to which the EEPROM contents should be copied.
`unsigned int`	`len`	The number of bytes to be read.

B.4.3 eep_buf_write()

Description:	This function writes a specified number of bytes to the EEPROM.	
Function:	`eep_buf_write()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned short`	`address`	The address (offset) inside the EEPROM to which data should be written.
`unsigned char`	`*buf`	A pointer to a data buffer containing the data to be written to the EEPROM.
`unsigned int`	`len`	The number of bytes to be written.

B.4.4 eep_total_size()

Description:	This function returns the total size of the EEPROM device.	
Function:	`eep_total_size()`	
Return:	`unsigned short`	The EEPROM size in bytes.
Parameters:	None.	

B.5 LEDs

B.5.1 led_init()

Description:	This function initializes the LED hardware.
Function:	`led_init()`
Return:	`void`
Parameters:	None.

B.5.2 led_set_all()

Description:	This function sets the three binary LEDs in one shot.
Function:	`led_set_all()`
Return:	`void`

Parameters:

Type	**Name**	**Description:**
`unsigned short`	`mask`	A bit mask representing the binary values of all three LEDs. Bit 0: Red Bit 1: Green Bit 2: Yellow 0 = Off, 1 = On

B.5.3 led_set()

Description:	This function turns on one LED.	
Function:	`led_set()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`position`	Specifies which LED should be turned on. 0: Red 1: Green 2: Yellow

B.5.4 led_clear()

Description:	This function turns off one LED.	
Function:	`led_clear()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`position`	Specifies which LED should be turned off. 0: Red 1: Green 2: Yellow

B.5.5 led_toggle()

Description:	This function toggles one LED.	
Function:	`led_toggle()`	
Return:	`void`	
Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`position`	Specifies which LED should be toggled. 0: Red 1: Green 2: Yellow

B.6 DIP Switches

B.6.1 switch_on()

Description:	This function returns the boolean value of a specified switch.	
Function:	`switch_on()`	

Return:	`boolean`	Boolean value of a switch.
		TRUE = ON
		FALSE = OFF

Parameters:		
Type	**Name**	**Description:**
`unsigned int`	`switch_num`	The logical number of the switch on the board. Note: on each board logical switch numbers are assigned. These do not necessarily match the physical switch number, so a physical to logical mapping scheme is provided for each board. Logical switch numbers allow an application to use one logical switch number across all boards.

Notes