TMS320VC547x ARM-Side Chip Support Library API Reference Guide

Preliminary

Literature Number: SPRU079
August 2002

TEXAS INSTRUMENTS

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Contents

Tables

Chapter 1

CSL Overview

This chapter introduces the Chip Support Library (CSL), briefly describes its architecture, and provides a generic overview of the collection of functions, macros, and constants that help you program ARM peripherals.

Topic **Page**

1.1 Introduction to CSL

The Chip Support Library(CSL) is a collection of functions, macros, and symbols used to configure and control on-chip peripherals. The goal is peripheral ease of use, shortened development time, portability, hardware abstraction, and some level of standardization and compatibility among TI devices.

1.1.1 How the CSL Benefits You

The benefits of the CSL include peripheral ease of use, shortened development time, portability, hardware abstraction, and a level of standardization and compatibility among devices. Specifically, the CSL offers:

- Standard Protocol to Program Peripherals

  The CSL provides you with a standard protocol to program on-chip peripherals. This protocol includes data types and macros to define a peripherals configuration, and functions to implement the various operations of each peripheral.

- Basic Resource Management

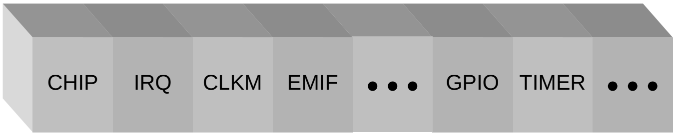
  Basic resource management is provided through the use of open and close functions for many of the peripherals. This is especially helpful for peripherals that support multiple channels.

1.1.2 CSL Architecture

The CSL consists of modules that are built and archived into a library file. Each peripheral is covered by a single module while additional modules provide general programming support.

Figure 1–1 illustrates the individual CSL modules. This architecture allows for future expansion because new modules can be added as new peripherals emerge.

Figure 1–1. CSL Modules



Although each CSL module provides a unique set of functions, some interdependency exists between the modules. For example, the TIMER module depends on the IRQ module because of TIMER interrupts; as a result, when you link code that uses the TIMER module, a portion of the IRQ module is linked automatically.

Each module has a compile-time support symbol that denotes whether or not the module is supported for a given device. For example, the symbol TIMER_SUPPORT has a value of 1 if the current device supports it and a value of 0 otherwise. The available symbols are located in Table 1–1. You can use these support symbols in your application code to make decisions.

Table 1–1 lists general and peripheral modules with their associated include file and the module support symbol that must be included in your application. The list also notes which modules are and are not supported by the CSL GUI.

Table 1–1. CSL Modules and Include Files

Peripheral Module (PER)	Description	Include File	Module Support Symbol
API	ARM port interface module	csl_api.h	API_SUPPORT
CHIP	General device module	csl_chip.h	CHIP_SUPPORT
CLKM	Clock manager module	csl_clkm.h	CLKM_SUPPORT
EIM	Ethernet interface module	csl_eim.h	EIM_SUPPORT
EMIF	External memory bus interface module	csl_emif.h	EMIF_SUPPORT
GPIO	Non-multiplexed general-purpose I/O module	csl_gpio.h	GPIO_SUPPORT
I2C	I^2C module	csl_i2c.h	I2C_SUPPORT
IRQ	Interrupt controller module	csl_irq.h	IRQ_SUPPORT
IRUART	IRDA UART module	csl_iruart.h	IRUART_SUPPORT
KBIO	Keyboard I/O module	csl_kbio.h	KBIO_SUPPORT
SDRAM	SDRAM interface module	csl_sdram.h	SDRAM_SUPPORT
SPI	Serial port interface module	csl_spi.h	SPI_SUPPORT
TIMER	Timer peripheral module	csl_timer.h	TIMER_SUPPORT
UART	UART module	csl_uart.h	UART_SUPPORT
WDTIM	Watchdog timer module	csl_wdtim.h	WDTIM_SUPPORT

Table 1–2 lists the 547x devices that the CSL supports and the ARM/Thumb libraries included in the CSL. The device support symbol must be used with the compiler (option -d), for the correct peripheral configuration to be used in your code.

Table 1–2. CSL Device Support

Library	**32-bit/16-bit**	**Endian**	**Device Support Symbol**
csl547xarm32.lib	32-bit	Big endian	CHIP_5470/CHIP_5471
csl547xarm32e.lib	32-bit	Little endian	CHIP_5470/CHIP_5471
csl547xarm16.lib	16-bit	Big endian	CHIP_5470/CHIP_5471
csl547xarm16e.lib	16-bit	Little endian	CHIP_5470/CHIP_5471

1.2 Naming Conventions

The following conventions are used when naming CSL functions, macros and data types.

Table 1–3. CSL Naming Conventions

Object Type	Naming Convention
Function	PER_funcName()†
Variable	PER_varName†
Macro	PER_MACRO_NAME†
Typedef	PER_Typename†
Function Argument	funcArg
Structure Member	memberName

† PER is the placeholder for the module name.

- ❑ All functions, macros and data types start with PER_ (where PER is the peripheral module name listed in Table 1–1) in uppercase letters.
- ❑ Function names use all lowercase letters. Uppercase letters are used only if the function name consists of two separate words. For example, PER_getConfig().
- ❑ Macro names use all uppercase letters; for example, TIMER_TCR_RMK.
- ❑ Data types start with an uppercase letter followed by lowercase letters. For example, TIMER_Handle.

1.3 CSL Data Types

The CSL provides its own set of data types that all begin with an uppercase letter. Table 1–4 lists the CSL data types as defined in the stdinc.h file.

Table 1–4. CSL Data Types

Data Type	Description
Bool	unsigned short
*PER*_Handle	void *
Int8	char
Int16	short
Int32	int
Uint8	unsigned char
Uint16	unsigned short
Uint32	unsigned int

1.4 CSL Functions

Table 1–5 provides a generic description of the most common CSL functions where *PER* indicates a peripheral module as listed in Table 1–1.

Note:

Not all of the peripheral functions are available for all the modules. See the specific module chapter for specific module information. Also, each peripheral module may offer additional peripheral specific functions.

The following conventions are used and are shown in Table 1–5:

- Italics indicate variable names.
- Brackets [...] indicate optional parameters.

CSL functions provide a way to program peripherals by:

- **Direct register initialization** using the PER_config() functions (see section 1.4.1).
- **Using functional parameters** using the PER_setup() function and various module specific functions (see section 1.4.2). This method provides a higher level of abstraction compared with the direct register initialization method, but typically at the expense of a larger code size and higher cycle count.

  Note: These functions are not available for all CSL peripheral modules.

Table 1–5. Generic CSL Functions

Function	Description
handle = *PER*_**open**(*channelNumber,* *flags*)	Opens a peripheral channel and then performs the operation indicated by *flags*; must be called before using a channel. The return value is a unique device handle to use in subsequent API calls.
*PER*_**config**(*[handle,]* **configStructure*)	Writes the values of the configuration structure to the peripheral registers. Initialize the configuration structure with: ❑ Integer constants ❑ Integer variables ❑ Merged field values created with the *PER_REG*_RMK macro
PER_**setup**(*[handle,]* *setupStructure)	Initializes the peripheral based on the functional parameters included in the initialization structure. Functional parameters are peripheral specific. This function may not be supported in all peripherals. Please consult the chapter that includes the module for specific details.

Table 1–5. Generic CSL Functions(Continued)

Function	Description
*PER*_**reset**(*[handle]* *)*	Resets the peripheral to its power-on default values.
*PER*_**close**(*handle*)	Closes a peripheral channel previously opened with *PER*_open(). The registers for the channel are set to their power-on defaults, and any pending interrupt is cleared.

1.4.1 Peripheral Initialization via Registers

The CSL provides a generic function to initialize the registers of a peripheral: *PER*_config() (where *PER* is the peripheral as listed in Table 1–1).

- ***PER*_config()** allows you to initialize a configuration structure with the appropriate register values and pass the address of that structure to the function, which then writes the values to the writable register. The CSL GUI uses this function to initialize peripherals. Example 1–1 shows an example of this method.

Example 1–1. Using PER_config or PER_configArgs

```
PER_Config myConfig = {
  reg0,
  reg1,
  …
};
main() {
…
PER_config(hPer, &myConfig);
…
}
```

1.4.2 Peripheral Initialization via Functional Parameters

The CSL also provides functions to initialize peripherals via functional parameters. This method provides a higher level of abstraction compared with the direct register initialization method, which produces larger code size and higher cycle count.

Even though each CSL module may offer different parameter–based functions, PER_setup() is the most commonly used. PER_setup() initializes the parameters in the peripheral that are typically initialized only once in your application. PER_setup() can then be followed by other module functions implementing other common run-time peripheral operations as shown in Example 1–2. Other parameter-based functions include module-specific functions such as GPIO_setDirection or UART_setBaudRate functions.

Example 1–2. Using PER_setup()

```
PER_setup mySetup = {param_1, .... param_n};


main() {
   ...
   PER_setup (hPer, &mySetup);
   ...
   }
```

1.5 CSL Macros

CSL macros to access registers and fields are being redefined. Please read the CSL readme file.

1.6 Resource Management and the Use of CSL Handles

CSL provides limited support for resource management in applications that involve multiple threads, reusing the same multi-channel peripheral device.

Resource management in CSL is achieved through calls to the PER_open and PER_close functions. The PER_open function normally takes a channel/port number as the primary argument and returns a pointer to a Handle structure that contains information about which channel/port was opened.

When given a specific channel/port number, the open function checks a global flag to determine its availability. If the port/channel is available, then it returns a pointer to a predefined Handle structure for this device.

If the device has already been opened by another process, then an invalid Handle is returned with a value equal to the CSL symbolic constant, INV.

Calling PER_close frees a port/channel for use by other processes. PER_close clears the in_use flag and resets the port/channel.

1.6.1 Using CSL Handles

CSL Handle objects are used to uniquely identify an opened peripheral channel/port or device. Handle objects must be declared in the C source, and initialized by a call to a PER_open function before calling any other API functions that require a handle object as argument. For example:

```
TIMER_Handle myTimer;  /* Defines a TIMER_Handle object, myTimer */
```

Once defined, the CSL Handle object is initialized by a call to PER_open:

```
.
.
myTimer = TIMER_open(TIMER_DEVO,TIMER_OPEN_RESET);  /* Open TIMER device 0 */
```

The call to TIMER_open initializes the handle, myTimer. This handle can then be used in calls to other API functions:

```
TIMER_start(myTimer);          /* Start the timer running */
.
.
.
TIMER_close(myTimer);           /* Free the TIMER device*/
```

Chapter 2

How To Use the 547x ARM-Side CSL

This chapter provides instructions on how to use the 547x ARM-side CSL libraries.

Topic **Page**

2.1 Using the CSL Libraries

The 547x ARM-side CSL is not integrated as part of the DSP/BIOS configuration tool. The 547x CSL is distributed in a library format (source and object libraries available).

Table 2–1 lists the 547x CSL directory structure.

Table 2–1. 547x ARM-Side CSL

CSL Component	Directory Structure
Library	c:\ti\tms470\csl\lib\
Source library	c:\ti\tms470\csl\lib\
Include files	c:\ti\tms470\csl\include\
Examples	c:\ti\examples\<target>\csl\
Documentation	c:\ti\docs

The process to use 547x ARM-Side CSL APIs is similar to other DSP CSLs:

Step 1: Header files to be included.

Include the *csl.h* and the *csl_per.h* for each peripheral (*per*) to be used. The include search path (shown in Table 2–1) is automatically set during the Code Composer Studio installation process.

Step 2: Specify the target device symbol to use by using either of the following:

- ❑ The –d CHIP_5470 or CHIP_5471 compiler command-line option.
- ❑ Selecting under Code Composer Studio: Project→Options and then under the Compiler Tab (Preprocessor), define the Symbol Field required (CHIP_5470 or CHIP_5471).

Step 3: Invoke the CSL initialization routine, CSL_init().

CSL_init() initializes any global variables and status information required.

Link with the corresponding CSL and RTS library depending on whether you are using the ARM (32-bit) or Thumb (16-bit) instruction set and the endianness.

See Table 2–2 for more information.

Table 2–2. CSL Library Models

Instruction Set	Endianness	CSL Library	RTS Library
ARM (32-bit)	Big Endian	csl547xarm32.lib	rts32.lib
ARM (32-bit)	Little Endian	csl547xarm32e.lib	rts32e.lib
Thumb (16-bit)	Big Endian	csl547xarm16.lib	rts16.lib
Thumb (16-bit)	Little Endian	csl547xarm16e.lib	rts16e.lib

Step 4: Determine if you must enable inlining.

Because some CSL ARM-Side functions are short (they may set only a single bit field), incurring the overhead of a C function call is not always necessary. If you enable inline, the CSL ARM-side declares these functions as *static inline*. Using this technique helps you improve code performance.

Example 2–1 illustrates these steps.

Example 2–1. Using the 547x ARM-Side CSL

```
#include <csl.h>
#include <csl_timer.h>
....
TIMER_Handle  myHandle;
main() {
    CSL_init();
    ...
    myHandle = TIMER_open (TIMER_DEV1, TIMER_OPEN_RESET);
    ....      // other TIMER APIs
    }
```

2.2 Rebuilding the CSL Library

We have provided a zipped archive of the CSL sources in the file csl_547x_src.zip. If the user needs to modify the CSL, it is recommended that s/he extract the sources from this zip-file and modify it before rebuilding the CSL.

An example project file (csl547xARM.pjt) is provided with the sources which may be useful for rebuilding the CSL library. The user may change the “Active Configuration” from the “Project Toolbar” to suit his instruction-set/endianness requirements before building the CSL library. However, there could be hard-coded path dependencies in the project file which the user is expected to resolve.

Chapter 3

API Module

The API module provides functions and macros for interfacing to and configuring the on-chip ARM Port Interface (API) module. The API interface, provides the ARM MCU access to a small portion of DSP memory through a 16-bit data path to the API RAM in the DSP sub-system. This on-chip shared API memory can be used by the ARM MCU to load code and boot-up the DSP core.

Topic **Page**

3.1 Overview

Table 3–1. API Descriptions

Syntax	Type	Description	Page
API_Config	S	API configuration structure.	3-3
API_Setup	S	API set-up structure.	3-3
API_config	F	Configures the API using the config structure.	3-4
API_getConfig	F	Gets API register values.	3-4
API_getEventId	F	Obtains the event ID for the API.	3-5
API_getMode	F	Returns the API mode.	3-5
API_getSetup	F	Gets the API configuration.	3-5
API_hostInterrupt	F	Checks for an active API interrupt from the DSP core.	3-6
API_interruptDsp	F	Generates an API interrupt on the DSP.	3-6
API_setup	F	Sets up the API using the set-up structure.	3-6

Note: F = Function; S = Structure

3.2 API Reference

API_Config — *API configuration structure*

Structure API_Config

Members

Uint32 wscr	API wait-state configuration register {API_REG}
Uint16 apcr	API control register {APIC}

Description This is the API configuration structure used to setup the ARM Port Interface. In order to configure the ARM Port Interface, the API_Config structure is initialized with the API register values and it's address is passed to the API_config function.

API_Setup — *API set-up structure*

Structure API_Setup

Members

Uint16 api_ws	wait state: no. of clock cycles API_DS is maintained low
Uint16 api_cs	hold-time: no. of clock cycles API_NRW is valid after release of API_DS
Uint16 api_bs	wait-state for back-to-back accesses (ex:32-bit)

Description This is the API setup structure which is used to configure the ARM Port Interface using the API_setup function. The API_Setup structure is to be initialized with the required parameters before calling the API_setup function.

API_config — *Configures the API using the configuration structure*

Function

```
void  API_config(
    API_Config    *config
)
```

Arguments

config   API configuration structure

Return Value

none

Description

Configures the ARM Port Interface using API device register values passed in through the API_Config structure

Example

```
API_Config myConfig = {
  0x00000093, // API Wait-State Configuration Register
  0x00000000  // API Control Register
};
...
API_config(&myConfig);
```

API_getConfig — *Gets API register values*

Function

```
void  API_getConfig(
    API_Config        *config
)
```

Arguments

config   API configuration structure

Return Value

none

Description

This function returns the values of the API registers in the API_Config structure provided by the user.

Example

```
API_Config myConfig;
API_getConfig(&myConfig);
```

API_getEventId — *Obtains the event ID for the API*

Function

```
Uint16  API_getEventId(
    void
)
```

Arguments none

Return Value Uint16

Description This function returns the event ID of the interrupt associated with the ARM Port Interface.

Example

```
Uint16 evt;
evt = API_getEventId( );
IRQ_enable(evt);
```

API_getMode — *Returns the API mode*

Function

```
Uint16  API_getMode(
    void
)
```

Arguments none

Return Value Uint16 — the API mode:

- API_MODE_SAM – shared access mode
- API_MODE_HOM – host only mode

Description Returns a value indicating whether the ARM Port Interface is in Host-Only Mode or Shared-Access Mode.

Example

```
if (API_getMode( ) == API_MODE_SAM) {
 //..//
}
```

API_getSetup — *Gets the API configuration*

Function

```
void  API_getSetup(
    API_Setup      *setup
)
```

Arguments setup — API set-up structure

Return Value none

Description This function returns the ARM Port Interface's current configuration in the API_Setup structure that is passed to it as argument.

Example

```
API_Setup mySetup;
API_getSetup(&mySetup);
```

API_hostInterrupt *Checks for an active API interrupt from the DSP core*

Function

```
Bool  API_hostInterrupt(
    void
)
```

Arguments none

Return Value Bool

Description The API host interrupt returns a boolean value indicating whether an interrupt from the DSP core is active.

Example

```
while(API_hostInterrupt( ));
```

API_interruptDsp *Generates an API interrupt on the DSP*

Function

```
void  API_interruptDsp(
    void
)
```

Arguments none

Return Value none

Description This function generates a DSP AINT/SIN9 interrupt on the DSP core.

Example

```
API_interruptDsp( );
```

API_setup *Sets up the API using the set-up structure*

Function

```
void  API_setup(
    API_Setup        *setup
)
```

Arguments setup   API set-up structure

Return Value none

Description Configure the ARM Port Interface using set-up parameters passed in through the API_Setup structure.

Example

```
API_Setup mySetup = {
  3, // API_WS
  2, // API_CS
  1  // API_BS
};
...
API_setup(&mySetup);
```

3.3 Register and Field Names

Table 3–2. API Module Register and Field Names

Register Name	Field Name(s)
API_WSCR	API_CS, API_BS, API_WS
API_APCR	HINT (R), DSPINT (W), APIMODE (R)

Note: R = Read only; W = Write only; fields not marked are R/W

Chapter 4

CHIP Module

The CHIP module contains code to perform chip-related and chip-specific functions. For the ARM, the CHIP module provides APIs for switching between modes, setting-up stacks, and hooking-up exception handlers.

Topic **Page**

4.1 Overview

Table 4–1. CHIP Descriptions

Syntax	Type	Description	Page
CHIP_getMode	F	Returns the current ARM mode.	4-3
CHIP_getSavedMode	F	Retrieves the mode from the SPSR.	4-4
CHIP_hookVector	F	Modifies an exception vector entry to branch to the specified handler.	4-4
CHIP_setMode	F	Changes to the specified ARM mode.	4-5
CHIP_setSavedMode	F	Changes the ARM mode in the SPSR.	4-5
CHIP_setupStack	F	Sets up the stack pointer for the specified mode.	4-6

Note: F = Function

4.2 API Reference

CHIP_getMode — *Returns the current ARM mode*

Function

```
Uint32  CHIP_getMode(
    void
)
```

Arguments none

Return Value

Uint32 — The current ARM mode:

- CHIP_MODE_USR
- CHIP_MODE_FIQ
- CHIP_MODE_IRQ
- CHIP_MODE_SVC
- CHIP_MODE_ABT
- CHIP_MODE_UND
- CHIP_MODE_SYS

Description

This function returns the current mode from the CPSR. Available ARM Modes are:

- CHIP_MODE_USR (user mode)
- CHIP_MODE_FIQ (FIQ mode)
- CHIP_MODE_IRQ (IRQ mode)
- CHIP_MODE_SVC (supervisor mod)
- CHIP_MODE_ABT (abort mode)
- CHIP_MODE_UND (undefined mode)
- CHIP_MODE_SYS (system mode)

See CHIP_setMode.

Example

```
if (CHIP_getMode() == CHIP_MODE_USR) {
  // do something
}
```

CHIP_getSavedMode *Retrieves the mode from the SPSR*

Function

```
Uint32  CHIP_getSavedMode(
    void
)
```

Arguments none

Return Value Uint32 The SPSR mode:

- CHIP_MODE_USR
- CHIP_MODE_FIQ
- CHIP_MODE_IRQ
- CHIP_MODE_SVC
- CHIP_MODE_ABT
- CHIP_MODE_UND
- CHIP_MODE_SYS

Description This function retrieves the ARM mode in the SPSR. Note that this function must be called from a privileged (non-USR) mode.

Example

```
int svdMod = CHIP_getSavedMode( );
```

CHIP_hookVector *Modifies exception vector entry to branch to specified handler*

Function

```
void  CHIP_hookVector(
    Uint32         exception,
    void          *handler
)
```

Arguments

exception The ARM exception to hook:

- CHIP_EXCP_RESET
- CHIP_EXCP_UNDEF
- CHIP_EXCP_SWI
- CHIP_EXCP_PREABT
- CHIP_EXCP_DATABT
- CHIP_EXCP_IRQ
- CHIP_EXCP_FIQ

handler Pointer to the handler function

Return Value none

Description This function hooks up an exception handler (‘dispatcher’) by modifying the specified exception’s top-level exception-vector.

Example

```
CHIP_hookVector (CHIP_EXCP_IRQ, &_IRQ_dispatcher);
```

CHIP_setMode *Changes to the specified ARM mode*

Function

```
Uint32  CHIP_setMode(
    Uint32          mode
)
```

Arguments

mode — The ARM mode to switch to:

- CHIP_MODE_USR
- CHIP_MODE_FIQ
- CHIP_MODE_IRQ
- CHIP_MODE_SVC
- CHIP_MODE_ABT
- CHIP_MODE_UND
- CHIP_MODE_SYS

Return Value

Uint32 — The previous ARM mode

Description

This function switches the ARM core to the specified mode by modifying the CPSR. For available ARM modes, see CHIP_getMode.

Example

```
oldMode = CHIP_setMode(CHIP_MODE_FIQ);
// do something //
CHIP_setMode(oldMode); // switch back mode
```

CHIP_setSavedMode *Changes the ARM mode in the SPSR*

Function

```
Uint32  CHIP_setSavedMode(
    Uint32          mode
)
```

Arguments

mode — The mode to set to in the SPSR:

- CHIP_MODE_USR
- CHIP_MODE_FIQ
- CHIP_MODE_IRQ
- CHIP_MODE_SVC
- CHIP_MODE_ABT
- CHIP_MODE_UND
- CHIP_MODE_SYS

Return Value

Uint32 — The old mode in the SPSR

Description

This function changes the ARM mode in the SPSR register so that the ARM switches to the specified mode when returning from within an exception handler. This function must be called from a privileged (non-USR) mode. For available ARM Modes see CHIP_getMode.

Example

```
int oldSvdMode = CHIP_setSavedMode(CHIP_MODE_SYS);
```

CHIP_setupStack *Sets up the stack pointer for the specified mode*

Function

```
void  CHIP_setupStack(
    Uint32          mode,
    void           *stack_pointer
)
```

Arguments

mode — The ARM mode to setup the stack for:
- CHIP_MODE_USR
- CHIP_MODE_FIQ
- CHIP_MODE_IRQ
- CHIP_MODE_SVC
- CHIP_MODE_ABT
- CHIP_MODE_UND
- CHIP_MODE_SYS

stack_pointer — The initial stack pointer

Return Value

none

Description

This function initializes the stack pointer for the specified ARM mode. The stack is assumed to be a “full descending” one and hence the ‘stack_pointer’ must point to the last word of the stack buffer. The function must be called from a privileged (non-USR) mode, if the current mode does not match the mode for which the stack is to be set up.

Example

```
Uint32 _stack_fiq[_STACK_SIZE_FIQ];
Uint32 _stack_irq[_STACK_SIZE_IRQ];
CHIP_setupStack(CHIP_MODE_FIQ, _stack_fiq + (sizeof(_stack_fiq)>>2) - 1);
CHIP_setupStack(CHIP_MODE_IRQ, _stack_irq + (sizeof(_stack_irq)>>2) - 1);
```

Chapter 5

CLKM Module

This module is in charge of controlling clock activity for the DSP, MCU, and peripherals. It includes configuration registers for DSP and MCU clock frequency programming. The clock module also manages the reset of all modules connected to the MCU.

Topic **Page**

5.1 Overview

Table 5–1. CLKM Descriptions

Syntax	Type	Description	Page
CLKM_Config	S	CLKM/PLL configuration structure.	5-3
CLKM_DEVICE_CNT	C	CLKM device count.	5-3
CLKM_OPEN_RESET	C	CLKM open reset flag.	5-3
CLKM_close	F	Closes previously opened CLKM device.	5-4
CLKM_config	F	Configures CLKM using configuration structure.	5-4
CLKM_getAudFrDivisor	F	Returns the audio clock frequency division factor.	5-5
CLKM_getConfig	F	Reads the current CLKM configuration values.	5-6
CLKM_getDspBootMode	F	Gets the DSP boot mode.	5-6
CLKM_open	F	Opens CLKM Device for use.	5-7
CLKM_reset	F	Resets the CLKM device.	5-7
CLKM_resetDsp	F	Resets hold/release the DSP or external peripherals.	5-8
CLKM_setAudClkFreq	F	Sets the audio clock frequency.	5-8
CLKM_setDspBootMode	F	Sets the DSP boot mode.	5-9
CLKM_switchClkMode	F	Switches to specified clock mode.	5-9

Note: C = Constant; F = Function; S = Structure

5.2 API Reference

CLKM_Config *CLKM/PLL configuration structure*

Structure CLKM_Config

Members

Uint32 clkmr	Clock configuration register
Uint32 dspr	DSP PLL register
Uint32 wkupr	Wake-up register
Uint32 audr	Audio register
Uint32 rstcr	Reset control register
Uint32 wdstr	Watchdog status register
Uint32 rstr	Reset register
Uint32 lpmr	Low-power mode register
Uint32 lpvr	Low-power value register
Uint32 pllccr	ARMSS PLL_REG clock control register

Description This is the CLKM configuration structure used to set up a CLKM device. User can create and initialize this structure and then pass its address to the CLKM_config function.

CLKM_DEVICE_CNT *CLKM device count*

Constant CLKM_DEVICE_CNT

CLKM_OPEN_RESET *CLKM open reset flag*

Constant CLKM_OPEN_RESET

Description This flag is used while opening CLKM device To open with reset; use CLKM_OPEN_RESET otherwise use 0.

Example See CLKM_open

CLKM_close — *Closes previously opened CLKM device*

Function

```
void  CLKM_close(
    CLKM_Handle       hClkm
)
```

Arguments

hClkm Device handle; see CLKM_open

Return Value

none

Description

Closes a previously opened CLKM device (see CLKM_open). The following tasks are performed:

- The CLKM event is disabled and cleared.
- The CLKM registers are set to their default values.

Example

```
CLKM_close(hClkm);
```

CLKM_config — *Configures CLKM using configuration structure*

Function

```
void  CLKM_config(
    CLKM_Handle       hClkm,
    CLKM_Config       *myConfig
)
```

Arguments

hClkm Device handle; see CLKM_open

myConfig Pointer to the configuration structure

Return Value

none

Description

Sets up the CLKM device using the configuration structure. The values of the structure variables are written to the CLKM registers.

Example

```
CLKM_Config MyConfig = {
    0x0u,
    0x000FFFFFu
    0x0u,
    0x0u,
    0x000FFFFFu,
    0x000FFFFFu
    0x0u,
   0x000FFFFFu
     0x0u,
    0x0u,
    0x000FFFFFu
};
...
CLKM_config(hClkm, &MyConfig);
```

CLKM_getAudFrDivisor *Returns the audio clock frequency division factor*

Function

Uint32 CLKM_getAudFrDivisor(
  CLKM_Handle hClkm
)

Arguments

hClkm Device Handle; see CLKM_open

Return Value

Uint32

Description

Returns the audio clock frequency division factor This will give the Audio clock frequency as per the equation:

AudioClkFreq = REF_CLK_FRQ / DivFactor.

where REF_CLK_FRQ is the reference clock frequency which depends on the board.

Example

```
Uint32 AudFrDiv;
AudFrDiv = CLKM_getAudFrDivisor(hClkm);
```

CLKM_getConfig *Reads the current CLKM configuration values*

Function

```
void  CLKM_getConfig(
    CLKM_Handle      hClkm,
    CLKM_Config      *config
)
```

Arguments

hClkm	Device Handle; see CLKM_open
config	Pointer to the destination configuration structure

Return Value none

Description Gets the current CLKM configuration structure

Example

```
CLKM_Config clkmCfg;
CLKM_getConfig(hClkm, &clkmCfg);
```

CLKM_getDspBootMode *Get the DSP boot mode*

Function

```
Uint16  CLKM_getDspBootMode(
    CLKM_Handle      hClkm
)
```

Arguments

hClkm	Device Handle; see CLKM_open

Return Value Uint16

Description Returns the DSP boot modes, possible modes are:

- CLKM_BOOT_API_MC – Microcontroller mode API memory
- CLKM_BOOT_API_MP – Microprocessor mode API memory
- CLKM_BOOT_ONCHIP – On chip RAM memory
- CLKM_BOOT_EXTMEM – External DSP memory

Example

```
Uint32 bMode;
bMode = CLKM_getDspBootMode(hClkm);
```

CLKM_open — *Opens CLKM device for use*

Function

```
CLKM_Handle  CLKM_open(
    Uint16          devNum,
    Uint16          flags
)
```

Arguments

devNum — Specifies the device to be opened

flags — Open flags
- CLKM_OPEN_RESET – resets the CLKM
- 0 – No reset

Return Value

CLKM_Handle — Device handle
INV – open failed

Description

Before a CLKM can be used, it must be 'opened' using this function. Once opened it cannot be opened again until it is 'closed' (see CLKM_close). The return value is a unique device handle that is used in subsequent CLKM API calls. If the open fails, 'INV' is returned.

If the CLKM_OPEN_RESET flag is specified, the CLKM module registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example

```
CLKM_Handle hClkm;
...
hClkm = CLKM_open(CLKM_DEV0, CLKM_OPEN_RESET);
```

CLKM_reset — *Resets the CLKM device*

Function

```
void  CLKM_reset(
    CLKM_Handle       hClkm
)
```

Arguments

hClkm — Device Handle; see CLKM_open

Return Value

none

Description

Resets the CLKM Device and sets the CLKM registers to their default values.

Example

```
CLKM_reset(hClkm);
```

CLKM_resetDsp — *Resets hold/release the DSP or external peripherals*

Function

```
void  CLKM_resetDsp(
    CLKM_Handle       hClkm,
    Uint16            flags
)
```

Arguments

hClkm	Device handle; see CLKM_open
flags	hold/reset flags

Return Value none

Description This API can be used to reset Hold/Release the DSP or external peripherals The flag denotes release/hold reset status of DSP or EXTERNAL PERIPHERALS or BOTH as mentioned below. The flags can be ORed.

- CLKM_RST_DSP_REL – Release DSP from reset
- CLKM_RST_EXT_REL – Release External peripherals from reset
- CLKM_RST_DSP_HLD – Hold DSP in reset
- CLKM_RST_EXT_HLD – Hold External Peripherals in reset

Example

```
CLKM_resetDsp(hClkm,CLKM_RST_DSP_REL);
```

CLKM_setAudClkFreq — *Sets the audio clock frequency*

Function

```
void  CLKM_setAudClkFreq(
    CLKM_Handle       hClkm,
    Uint32            AudFrDiv
)
```

Arguments

hClkm	Device Handle; see CLKM_open
AudFrDiv	Audio frequency divisor

Return Value none

Description Sets the audio clock frequency User has to supply the audio frequency divisor, by using the equation:

AudFrDiv = REF_CLK_FRQ / AudioClkFreq

Example

```
CLKM_setAudClkFreq(hClkm, 12);
```

CLKM_setDspBootMode *Sets the DSP boot mode*

Function

```
void  CLKM_setDspBootMode(
    CLKM_Handle      hClkm,
    Uint16           bMode
)
```

Arguments

hClkm	Device handle; see CLKM_open
bMode	DSP boot mode

Return Value none

Description Sets the DSP boot mode, possible modes are:

- CLKM_BOOT_API_MC – Microcontroller mode API memory
- CLKM_BOOT_API_MP – Microprocessor mode API memory
- CLKM_BOOT_ONCHIP – On chip RAM memory
- CLKM_BOOT_EXTMEM – External DSP memory

Example

```
CLKM_setDspBootMode(hClkm, CLKM_BOOT_EXTMEM);
```

CLKM_switchClkMode *Switches to specified clock mode*

Function

```
void  CLKM_switchClkMode(
    CLKM_Handle      hClkm,
    Uint16           mode
)
```

Arguments

hClkm	Device handle; see CLKM_open
mode	Clock mode

Return Value none

Description Switches the modes of the ARM subsystem clock. Available modes are:

- CLKM_MOD_LOW_POW
- CLKM_MOD_NORMAL
- CLKM_MOD_DIV2
- CLKM_MOD_DIV4

Note

1) Low-power mode (CLKM_MOD_LOW_POW): First make sure that the low-power mode counter is bigger than 512 before switching to divide by 512 mode.

2) Normal mode (CLKM_MOD_NORMAL): It is possible to stop the low_power_freq clock by writing a ‘0’ inside CLKM_LPVR but this should be done after some delays since several stages of synchronization are implemented in hardware. Don’t do immediately after CLKM_LPVR = 0.

3) Divide by 2 or 4 mode(CLKM_MOD_DIV2 or CLKM_MOD_DIV4): It is FORBIDDEN to switch to divide by 2 mode using the test mode. Switching to divide by 2 mode should be done using the normal mode; i.e., through PLL register setting.

Example

```
CLKM_switchClkMode(hClkm,CLKM_MOD_LOW_POWER);
```

5.3 Register and Field Names

Table 5–2. CLKM Module Register and Field Names

Register Name	Field Name(s)
CLKM_CLKMR	BLKCLKSTOP
CLKM_DSPR	MPNMC, APIBN, DPLLSHUTOFF, DPLLFRRSN, DPLLFRPLLDIVN, DPLLFRONOFF, DPLLFRDIV, DPLLFRDIVN
CLKM_WKUPR	BLKINTRWKUP
CLKM_AUDR	AUDIOCLKCMP
CLKM_RSTCR	EXTNRST, DSPNRST
CLKM_WDSTR	WDSTATUS
CLKM_RSTR	RESET
CLKM_LPMR	DIV2, LOWPOW
CLKM_PLLCCR	PLLMUL, PLLDIV, PLLCNT, PLLONOFF, PLLNDIV, STATUS
CLKM_LPVR	LOWPOWCNT

Note: All fields are Read/Write

Chapter 6

EIM Module

EIM module provides a straightforward and effective method of integrating an IEEE802.3/Ethernet MAC functionality onto a processor IO subsystem. The EIM module of CSL provides functions and macros for configuration EIM/ EMAC module in c5471x device.

Topic **Page**

6.1 Overview

Table 6–1. EIM Descriptions

Syntax	Type	Description	Page
EIM_Setup	S	EIM set-up structure.	6-3
EIM_EnetSyserr	S	ENET0 system error structure	6-5
EIM_Stats	S	EIM status structure	6-6
EIM_clearPhyIntr	F	Clears PHY interrupt.	6-6
EIM_clearStats	F	Clears global stats parameters.	6-6
EIM_close	F	Closes a previously opened ENET port.	6-7
EIM_disablePhy	F	Powers down PHY and isolates PHY from MII.	6-7
EIM_eimIsr	F	Default EIM interrupt.	6-7
EIM_flowCtrlEnable	F	Enables flow control.	6-8
EIM_flowCtrlDisable	F	Disables flow control.	6-8
EIM_getEventId	F	Returns EIM interrupt event ID.	6-8
EIM_getPhyEventId	F	Returns PHY interrupt event ID.	6-9
EIM_getStats	F	Gets the current device statistics.	6-9
EIM_intrConfig	F	Configures EIM interrupt based on passed-in configuration structure.	6-9
EIM_initPhy	F	Uses Management Data Interface to configure PHY.	6-10
EIM_open	F	Opens the EIM peripheral at the given port number.	6-10
EIM_phyIntrConfig	F	Configures PHY interrupt.	6-11
EIM_phyIsr	F	Sets up the EMAC register.	6-11
EIM_receivePacket	F	Receives a data packet from EIM packet memory.	6-12
EIM_setup	F	Configures the EIM module.	6-13
EIM_sendPacket	F	Sends the data packet pointed to by pBuffer.	6-14

Note: F = Function; S = Structure

6.2 API Reference

EIM_Setup *EIM set-up structure*

Structure

Uint8	phyMode
Bool	phyLoopback
Bool	phyColTest
Uint8	mWidth
Uint32	macAddrHigh
Uint32	macAddrLow
Uint16	bufSize
Uint32	numEnetTxDesc
Uint32	numEnetRxDesc
Uint8	rxMode
Uint32	logicAddrFiltHigh
Uint32	logicAddrFiltLow
Uint8	rxThreshold
Bool	rejectSfe
Bool	Loopback
Bool	FDLoopback
Uint16	backoffSeed
Uint16	vtype

Members

phyMode — PHY mode. Valid symbolic values are:
- PHY_HALF_10 – 0x0
- PHY_FULL_10 – 0x1
- PHY_HALF_100 – 0x2
- PHY_FULL_100 – 0x3
- PHY_AUTONEGOTIATE – 0x4

phyLoopback — PHY loop back. Valid symbolic values are:
- PHY_LOOPBACK – 1
- PHY_NO_LOOPBACK – 0

phyColTest — PHY Collision test enable. Valid symbolic values are:
- PHY_COLTEST – 1
- PHY_NO_COLTEST – 0

mWidth — Width of MII port (serial mode or Nibble mode). Valid symbolic values are:
- PHY_NIBBLE_MODE – 1
- PHY_NO_NIBBLE_MODE – 0

macAddrHigh	MAC address bits 32–48
macAddrLow	MAC address bits 0–32
bufSize	packet buffer size. The bufSize has to be set with a multiple of 4 and a minimum of 64 (and <=1536).
numEnetTxDesc	number of ENET TX descriptors. Valid value should satisfy following: (numEnetTxDesc + numEnetRxDesc) * (8+bufSize +4) <= 16 K bytes
numEnet0RxDesc	number of ENET RX descriptors. Valid value should satisfy following: (numEnetTxDesc + numEnetRxDesc) * (8+bufSize +4) <= 16 K bytes
rxMode	Addressing modes. Valid symbolic values are: ❑ ENET_ADR_PROMISCUOUS – 0x08 ❑ ENET_ADR_BROADCAST – 0x04 ❑ ENET_ADR_LOGICAL – 0x02 ❑ ENET_ADR_PHYSICAL – 0x01
logicAddrFiltHigh	logic address Hash Filter register bits 63:32
logicAddrFiltLow	logic address Hash Filter register bits 31:0
rxThreshold	Number of pending RX descriptors to reach in an ENET ring to trigger the TX flow control frame on this ENET
rejectSfe	Reject short frame error. Valid symbolic values are: ❑ ENET_RJCT_SFE – 1 ❑ ENET_NO_RJCT_SFE – 0
loopback	MAC loop back. Valid symbolic values are: ❑ ENET_LOOPBACK – 1 ❑ ENET_NO_LOOPBACK – 0

FDLoopback	Full-duplex wrap. When both FDLoopback and loopback are set to 1, the packet reception from network will be stopped. Valid symbolic values are: ❑ ENET_FD_LOOPBACK – 1 ❑ ENET_NO_FD_LOOPBACK – 0
backoffSeed	Backoff seed setup and backoff retry setup.
vtype	Virtual LAN tag.

Note: Limitations

Parameters fixed at initialization: These parameters are fixed when we initialize EIM, but they always can be changed by directly access the registers if needed through CSL MACROS.

- ❑ EIM_MODE_E0 bit FIFO_EN = 0 FIFO access disabled (normal mode)
- ❑ EIM_MODE_E0 bit DPNET = 0 normal mode
- ❑ EIM_FLW_CNTRL_E0 = 0 flow control could be enabled by using function EIM_flowControlEnable

EIM_EnetSyserr

Structure

```
Uint32    txFrameError
Uint32    rxByteCountError
Uint32    txFifError
Uint32    rxOverflowError
Uint32    txUnderflowError
```

Members

txFrameError	ENET system error for transmit byte count or PAD_CRC improperly set.
rxByteCountError	ENET system error for receive byte count error
txFifError	ENET system error for transmit First-in-frame error
rxOverflow	ENET system error for receive buffer memory overflow
txUnerflow	ENET system error for transmit buffer memory underflow

EIM_Stats

Structure

Uint32	txHangCnt
Uint32	rxHangCnt
Uint32	rxCheckHangFlag
Uint32	resetCnt
Uint32	linkChangeCnt
EIM_enetSyserr	enetErr

Members

txHangCnt	Transmit hang counter
rxHangCnt	receiver hang counter
rxCheckHangFlag	receiver check hang flag
resetCnt	reset EIM counter.
linkChangeCnt	PHY link change counter
enetErr	ENET system error structure

EIM_clearPhyIntr *Clears PHY interrupt*

Function void EIM_clearPhyIntr()

Arguments none

Return Value none

Description This function clears PHY interrupt.

Example

```
EIM_clearPhyIntr();
```

EIM_clearStats *Clears global stats parameters*

Function void EIM_clearStats()

Arguments none

Return Value none

Description This function clears global stats parameters.

Example

```
EIM_clearStats();
```

EIM_close

Closes a previously opened ENET port

Function void EIM_close(EIM_Handle hEim)

Arguments hEim Device handle (see EIM_open)

Return Value none

Description Closes a previously opened ENET port. The EIM registers are set to their default values and any associated interrupts are disabled and cleared.

Example

```
EIM_Handle thisEim;
...
EIM_close(thisEim);
```

EIM_disablePhy

Powers down PHY and isolates PHY from MII

Function void EIM_disablePhy()

Arguments none

Return Value none

Description This function powers down PHY and isolates PHY from MII.

Example

```
EIM_disablePhy();
```

EIM_eimIsr

Default EIM interrupt

Function void EIM_eimIsr()

Arguments none

Return Value none

Description This function is default EIM interrupt function. It handles the ENET system errors and updated the global stats parameters.

Example see EIM_intrConfig

EIM_flowCtrlEnable *Enables flow control*

Function void EIM_flowCtrlEnable(Uint16 flowCtrlMask)

Arguments flowCtrlMask Flow control register set-up mask. Valid symbolic values are:

- ENET_FLCNT_BACK_PSR – 0x02
- ENET_FLCNT_RX_FLWEN – 0x01
- ENET_FLCNT_DISABLE – 0x00

Return Value none

Description This function enables flow control based on passed-in flow control mask (flow control is disabled in EIM_setup()).

Example

```
Uint16 flMask = ENET_FLCNT_RX_FLWEN;
...
EIM_flowCtrlMask(flMask);
```

EIM_flowCtrlDisable *Disables flow control*

Function void EIM_flowCtrlDisable()

Arguments none

Return Value none

Description This function disables flow control.

Example

```
EIM_flowCtrlDisable();
```

EIM_getEventId *Returns EIM interrupt event ID*

Function Uint32 EIM_getEventId()

Arguments none

Return Value EIM interrupt ID

Description Returns EIM interrupt event ID.

Example

```
Uint32 eventID;
...
eventID = EIM_getEventID();
```

EIM_getPhyEventId *Returns PHY interrupt event ID*

Function Uint32 EIM_getPhyEventId()

Arguments none

Return Value PHY interrupt ID

Description Returns PHY interrupt event ID.

Example

```
Uint32 phyEventID;
...
phyEventID = EIM_getEventID();
```

EIM_getStats *Gets the current device statistics*

Function EIM_pStats EIM_getStats();

Arguments

Return Value A pointer to the statistics information.

Description Called to get the current device statistics. The statistics structure contains a collection of event counts for various packet sent and receive properties.

Example

```
EIM_pStats currentStats;
...
currentStats = EIM_getStats();
```

EIM_intrConfig *Configures EIM interrupt based on passed-in configuration structure*

Function void EIM_intrConfig(IRQ_Config *eimIrqConfig, Uint16 eimIntrRegMask)

Arguments

eimIrqConfig	a pointer to IRQ_Config structure. If NULL is passed, a default structure will be used and EIM_eimIsr() is used as interrupt function.
EimIntrRegMask	EIM interrupt register mask. If NULL is passed, a default value will be used.

Return Value none

Description This function configures the EIM interrupt based on the passed-in configuration structure. If a NULL is passed in, the default configuration will be used to configure EIM interrupt.

Example

```
...
EIM_IntrConfig(NULL, 0);
```

EIM_initPhy *Uses Management Data Interface to configure PHY*

Function Uint8 EIM_initPhy(U8 phyMode, Bool phyLoopback, Bool phyColTest)

Arguments

phyMode PHY mode. It could be one of the following values:

- HALF–10 = 0
- FULL–10 = 1
- HALF–100 = 2
- FULL–100 = 3
- AUTONEGOTIATE = 4

phyLoopback PHY loop back setup

phyColTest PHY Collision test enable

Return Value PHY Mode as follows:

- PHY_HALF–10 – 0
- PHY_FULL–10 – 1
- PHY_HALF–100 – 2
- PHY_FULL–100 – 3
- PHY_FAIL_WRONG_PHYID – 4
- PHY_FAIL_LINKDOWN – 5
- PHY_FAIL_INVALID_PHYMODE – 6
- PHY_FAIL_CHECKDUPLEX_ERROR – 7

Description This function uses Management Data Interface to configure PHY.

Example

```
Uint8       tPhyMode = FULL-100
Bool         tPhyLoopback = 0;
Bool         tPhyColtest = 0;
Uint8        phyReturn;
...
phyReturn    EIM_initPhy(tPhyMode, tPhyLoopback, tPhyColtest);
```

EIM_open *Opens the EIM peripheral at the given port number*

Function EIM_Handle EIM_open(Uint16 devNum, Uint16 flags)

Arguments

devNum Port number to open EIM ENET. Valid symbolic values are:

- EIM_DEV0 – 0

flags — EIM reset flag. Valid symbolic values are:
- EIM_OPEN_RESET – 0x00000001
- EIM_OPEN_NO_RESET – 0x00000000

Return Value The function returns a handle that is used in most EIM functions calls.

Description Opens the EIM peripheral at the given port number (in 5471 only one ENET). Before a EIM can be used, it must be opened by this function. Once opened, it cannot be open again until it is closed (see EIM_close). The return value is a unique device handle that is used in subsequent EIM API calls.

Example

```
Uint16      tDevNum = EIM_DEV0;
Uint16       tFlag = EIM_OPEN_NO_FLAG;
...
EIM_open(tDevNum, tFlag);
```

EIM_phyIntrConfig *Configures PHY interrupt*

Function void EIM_phyIntrConfig(IRQ_Config *phyIrqConfig, Uint32 mdIntrRegMask)

Arguments

phyIrqConfig — A pointer to an IRQ_Config structure. If NULL is passed, a default structure will be used and EIM_phyIsr() is used as an interrupt function.

mdIntrRegMask — PHY interrupt register mask. If NULL is passed, a default value will be used.

Return Value none

Description This function configures the PHY interrupt based on the passed-in configuration structure. If a NULL is passed in, default configuration will be used to configure PHY interrupt.

Example

```
EIM_phyIntrConfig(NULL, 0);
```

EIM_phyIsr *Sets up the EMAC register*

Function void EIM_phyIsr()

Arguments none

Return Value none

Description This function is the PHY interrupt (link change) function, it sets up the EMAC register for the PHY link change.

Example See EIM_phyIntrConfig

EIM_receivePacket *Receives a data packet from EIM packet memory*

Function

Int16 EIM_receivePacket(EIM_Handle hEIM, Uint32 *pPacket, EIM_Setup *params);

Arguments

pPacket — Pointer to the data buffer which will contain received data (the data buffer size should be great than 1518 byes to avoid data overflow.

hEim — Handle to ENET port obtained by EIM_open().

params — Pointer to an initialized configuration structure.

Return Value

RXP_ERROR = –1 means there is an error in received packet.

RXP_NO_PACKETS = 0 means not received any packet data. Otherwise, this function returns the length of received data in bytes.

Description

Receives a data packet from EIM packet memory.

Example

```
EIM_Handle      tHEIM;
Uint32 * tpPacket;
EIM_Setup tParams {  PHY_FULL_100,
                     PHY_LOOPBACK,
                     PHY_NO_COLTEST,
                     PHY_NIBBLE_MODE,
                     CPU_ADDRES_HI,
                     CPU_ADDRESS_LO,
                     128,
                     58,
                     59,
                     (ENET_ADR_PROMISCUOUS |
                     ENET_ADR_BROADCAST |
                     ENET_ADR_PHYSICAL),
                     0x00000000,
                     0x00000000,
                     24,
                     ENET_NO_RJCT_SFE,
                     ENET_NO_LOOPBACK,
                     ENET_NO_FD_LOOPBACK,
                     0,
                     ENET_VTYPE
};
Int16           rcvFlag;
rcvFlag = EIM_receiveFlag(tHEIM, tpPacket, &tParams);
```

EIM_setup *Configures the EIM module*

Function

Uint8 EIM_setup (EIM_Handle hEim, Bool phyReset, EIM_Setup *params,)

Arguments

params pointer to an initialized set-up structure

PhyReset specify if PHY will be reset or not

- PHY_RESET – 1
- PHY_NO_RESET – 0

hEim Handle to ENET port obtained by EIM_open().

Return Value

Error code as following:

- PHY_HALF–10 – 0
- PHY_FULL–10 – 1
- PHY_HALF–100 – 2
- PHY_FULL–100 – 3
- PHY_FAIL_WRONG_PHYID – 4
- PHY_FAIL_LINKDOWN – 5
- PHY_FAIL_INVALID_PHYMODE – 6
- PHY_FAIL_CHECKDUPLEX_ERROR – 7
- PHY_NO_RESET – 8

Description

This function configures the EIM module based on params passed in (setup EIM registers, initialize packet memory and does PHY init if required).

Example

```
EIM_Handle      tHEIM;
Bool             tPhyReset = PHY_RESET;
EIM_Setup tParams {  PHY_FULL_100,
                     PHY_LOOPBACK,
                     PHY_NO_COLTEST,
                     PHY_NIBBLE_MODE,
                     CPU_ADDRES_HI,
                     CPU_ADDRESS_LO,
                     128,
                     58,
                     59,
                     (ENET_ADR_PROMISCUOUS |
                     ENET_ADR_BROADCAST |
                     ENET_ADR_PHYSICAL),
                     0x00000000,
                     0x00000000,
                     24,
                     ENET_NO_RJCT_SFE,
                     ENET_NO_LOOPBACK,
                     ENET_NO_FD_LOOPBACK,
                     0,
                     ENET_VTYPE
};
Int16           rcvFlag;
rcvFlag = EIM_receiveFlag(tHEIM, tpPacket, &tParams);
```

EIM_sendPacket *Sends the data packet pointed to by pBuffer*

Function

Uint8 EIM_sendPacket (EIM_Handle hEim , Uint32 *pPacket, Int16 pktLen, EIM_Setup, *params)

Arguments

pPacket	Pointer to the data buffer which needs to be send out
PktLen	Length of data buffer in bytes
hEim	Handle to ENET port obtained by EIM_open()
params	Pointer to an initialized configuration structure

Return Value

TXP_OVERFLOW = 0 means no descriptor buffer available

TXP_PASS = 1 successfully puts the data into packet memory

Description

Sends the data packet pointed to by pBuffer. This function will eventually check descriptor ownership, get buffers, copy data, send packets.

Example

```
EIM_Handle      tHEIM;
Uint32 * tpPacket;
EIM_Setup tParams {  PHY_FULL_100,
                     PHY_LOOPBACK,
                     PHY_NO_COLTEST,
                     PHY_NIBBLE_MODE,
                     CPU_ADDRES_HI,
                     CPU_ADDRESS_LO,
                     128,
                     58,
                     59,
                     (ENET_ADR_PROMISCUOUS |
                     ENET_ADR_BROADCAST |
                     ENET_ADR_PHYSICAL),
                     0x00000000,
                     0x00000000,
                     24,
                     ENET_NO_RJCT_SFE,
                     ENET_NO_LOOPBACK,
                     ENET_NO_FD_LOOPBACK,
                     0,
                     ENET_VTYPE
};
Int16           tPktLen = 128;
Int16           txFlag:
txFlag = EIM_sendPacket(tHEIM, tpPacket, pktLen, &tParams);
```

6.3 Register and Field Names

Table 6–2. EIM Module Register and Field Names

Register Name	Field Name(s)
EIM_CTRL	ESMEN, CPU_ENET0_EN, ENET0_FLWCNTEN, ENET0_RXEN, ENET0_TXEN, CPU_RXEN, CPU_TXEN
EIM_STATUS	CPU_TX_LIF (RC), CPU_RX_LIF (RC), CPU_TX (RC), CPU_RX (RC), ENET0_ERR (RC), ENET0_TX (RC), ENET0_RX (RC)
EIM_CPUTXBA	TXCPU_BA
EIM_CPURXBA	RXCPU_BA
EIM_BUFSIZE	BUFSIZE
EIM_FILTER	MCLAEN, LOGICALEN, MULTICASTEN, BROAD-CASTEN, DAEN
EIM_CPUDA_1	DAR_1
EIM_CPUDA_0	DAR_0 MSW, DAR_0 LSW
EIM_MFV_1	MFV_1
EIM_MFV_0	MFV_0
EIM_MFM_1	MFM_1
EIM_MFM_0	MFM_0
EIM_RXTH	RXTH
EIM_RX_CPU_RDY	CPURX_RDY (W)
EIM_INT_EN	CPU_TX_LIF, CPU_RX_LIF, CPU_TX, CPU_RX, ENET0_ERR, ENET0_TX, ENET0_RX
EIM_ENET0_TX_DESC	ENET0_TX_PTR (R)
EIM_ENET0_RX_DESC	ENET0_RX_PTR (R)
EIM_CPU_TX_DESC	CPU_TX_PTR (R)
EIM_CPU_RX_DESC	CPU_RX_PTR (R)
EIM_MODE_E0	FIFO_EN, RJCT_SFE, DPNET, MWIDTH, WRAP, FDWRAP, DUPLEX, ENABLE

Note: R = Read only; C = Cleared after read; W = Write only; fields not marked are R/W

Table 6–2. EIM Module Register and Field Names (Continued)

Register Name	Field Name(s)
EIM_NEW_RBOF_E0	HALT_RBO, NEW_RBOF
EIM_RBOF_CNT_E0	RBOF_CNT (R)
EIM_FLW_CNT_E0	FLW_CNT
EIM_FLW_CNTRL_E0	BACK_PSR, RX_FLW_EN
EIM_VTYPE_E0	VTYPE
EIM_SE_SR_E0	TX_FE, RX_BCE, FIFE, OFLW, UFLW
EIM_TX_BUF_RDY_E0	
EIM_TDBA_E0	TDBA
EIM_RDBA_E0	RDBA
EIM_PAR1_E0	PAR_1
EIM_PAR0_E0	PAR_0 MSW, PAR_0 LSW
EIM_LAR1_E0	LAR_1
EIM_LAR0_E0	LAR_0
EIM_ADR_MODE_E0	ESAC, EBAC, ELAC, EPAC
EIM_DRP_E0	DRPC
EIM_MODE_E0	ENABLE, DUPLEX, FDWRAP, WRAP, MWIDTH, DPNET, RJCT_SFE, FIFO_EN
EIM_NEW_RBOF_E0	NEW_RBOF, RTRY, HALT_RBO
EIM_RBOF_CNT_E0	RBOF_CNT
EIM_FLW_CNT_E0	FLW_CNT
EIM_FLW_CNTRL_E0	RX_FLW_EN, BACK_PSR
EIM_VTYPE_E0	VTYPE
EIM_SE_SR_E0	UFLW, OFLW, FIFE, RX_BCE, TX_FE
EIM_TX_BUF_RDY_E0	
EIM_TDBA_E0	TDBA
EIM_RDBA_E0	RDBA

Note: R = Read only; C = Cleared after read; W = Write only; fields not marked are R/W

Table 6–2. EIM Module Register and Field Names (Continued)

Register Name	Field Name(s)
EIM_PAR1_E0	PAR_1
EIM_PAR0_E0	PAR_0 LSW, PAR_0 MSW
EIM_LAR1_E0	LAR_1
EIM_LAR0_E0	LAR_0
EIM_ADR_MODE_E0	EPAC, ELAC, EBAC, ESAC
EIM_DRP_E0	DPRC

Note: R = Read only; C = Cleared after read; W = Write only; fields not marked are R/W

Chapter 7

EMIF Module

The EMIF module provides functions and macros for interfacing to and configuring the on-chip External Memory Interface (EMIF) module. The EMIF is used to interface the MCU to external ARM memories by providing the necessary control signals, as well as address and data management to the external buses. External memory devices supported are ROM (Flash), SRAM and SDRAM. The memory interface provides support for 8-, 16- and 32-bit wide ROM (Flash) and SRAM memories.

Topic **Page**

7.1 Overview

Table 7–1. EMIF Descriptions

Syntax	Type	Description	Page
EMIF_Config	S	EMIF configuration structure.	7-3
EMIF_Setup	S	EMIF set-up structure.	7-3
EMIF_config	F	Configures the EMIF using the config structure.	7-4
EMIF_getConfig	F	Retrieves the EMIF register values.	7-4
EMIF_getSetup	F	Get the EMIF configuration.	7-5
EMIF_setup	F	Sets up the EMIF using the set-up structure.	7-5

Note: F = Function; S = Structure

7.2 API Reference

EMIF_Config — *EMIF configuration structure*

Structure EMIF_Config

Members

Uint32 cs0r

Uint32 cs1r

Uint32 cs2r

Uint32 cs3r

Uint32 cs4r

Uint32 bscr

Description This is the EMIF configuration structure used to set up the External Memory Interface. In order to configure the EMIF, the EMIF_Config structure is initialized with the EMIF register values and its address is passed to the EMIF_config function.

EMIF_Setup — *EMIF set-up structure*

Structure EMIF_Setup

Members

Uint16 waitState	no. of wait-states
Uint16 deviceSize	device size: 8/16/32 bit
Bool writeEnable	writable memory
Bool endianness	little or big endian
Uint16 dummyCycles	dummy cycles to be inserted during bank switching
Uint16 dummyCyclesMatrix	bank-switching dummy-cycle matrix
Uint16 addlWriteWS	wait-state insertion during write

Description This is the EMIF set-up structure which is used to configure an External Memory Interface chip-select line using the EMIF_setup function. The EMIF_Setup structure is to be initialized with the required parameters before calling the EMIF_setup function.

EMIF_config *Configures the EMIF using the config structure*

Function

```
void  EMIF_config(
    EMIF_Config       *config
)
```

Arguments config the initialized EMIF configuration structure

Return Value none

Description Configures the External Memory Interface using EMIF device register values passed in through the EMIF_Config structure.

Example

```
EMIF_Config myConfig = {
  0x00000151, // CS0_REG
  0x00000151, // CS1_REG  0x00000151, // CS2_REG
  0x00000151, // CS3_REG
  0x00000151, // CS4_REG
  0x00000000  // BS_CONFIG
};
EMIF_config(&myConfig);
```

EMIF_getConfig *Retrieves the EMIF register values*

Function

```
void  EMIF_getConfig(
    EMIF_Config       *config
)
```

Arguments config EMIF configuration structure

Return Value none

Description The function retrieves the current values of the EMIF device registers in the passed-in EMIF “config structure.”

Example

```
EMIF_Config myConfig;
EMIF_getConfig(&myConfig);
```

EMIF_getSetup — *Gets the EMIF configuration*

Function

```
void EMIF_getSetup(
    Uint16          eChipSelect,
    EMIF_Setup      *setup
)
```

Arguments

eChipSelect — Chip select line to be configured
- EMIF_CS0
- EMIF_CS1
- EMIF_CS2
- EMIF_CS3
- EMIF_CS4

setup — EMIF setup structure

Return Value none

Description This function returns the External Memory Interface's current configuration in the EMIF_Setup structure that is passed to it as argument.

Example

```
EMIF_Setup mySetup;
EMIF_getSetup(EMIF_CS0, &mySetup);
```

EMIF_setup — *Sets up the EMIF using the set-up structure*

Function

```
void EMIF_setup(
    Uint16          eChipSelect,
    EMIF_Setup      *setup
)
```

Arguments

eChipSelect — Chip select line to be configured
- EMIF_CS0
- EMIF_CS1
- EMIF_CS2
- EMIF_CS3
- EMIF_CS4

setup — the initialized EMIF setup structure

Return Value none

Description Configures the External Memory Interface using set-up parameters passed in through the EMIF_Setup structure.

Example

```
EMIF_Setup mySetup = {
  1, // wait-states
  EMIF_DEVICESIZE_16, // device-size
  EMIF_READWRITE, // write-enable
  EMIF_ENDIAN_LITTLE, // little endian
  1, // dummy-cycles
  0, // dummy-cycle matrix
  0 // no wait-state insertion during write
};
EMIF_setup(EMIF_CS0, &mySetup);
```

7.3 Register and Field Names

Table 7–2. EMIF Module Register and Field Names

Register Name	Field Name(s)
EMIF_CS0R	WS1, DC, BIGEND, WE, DVS, WS
EMIF_CS1R	WS1, DC, BIGEND, WE, DVS, WS
EMIF_CS2R	WS1, DC, BIGEND, WE, DVS, WS
EMIF_CS3R	WS1, DC, BIGEND, WE, DVS, WS
EMIF_CS4R	WS1, DC, BIGEND, WE, DVS, WS
EMIF_BSCR	SDRAM, CS

Note: All fields are Read/Write

Chapter 8

GPIO Module

This module provides 20 general-purpose I/Os and 16 Keyboard I/Os, configurable in read or write mode by internal registers. The 16 I/Os of KBGPIO can also be used as normal GPIO pins. Each GPIO is associated with six configuration/status bits. The configuration/status bits are accessible through 12 memory-mapped registers.

Topic **Page**

8.1 Overview

Table 8–1. GPIO Descriptions

Syntax	Type	Description	Page
GPIO_Config	S	GPIO configuration structure.	8-3
GPIO_DEVICE_CNT	C	GPIO device count.	8-3
GPIO_OPEN_RESET	C	GPIO open reset flag.	8-3
GPIO_Setup	S	GPIO set-up structure.	8-4
GPIO_clearPinDelta	F	Clears pin outputs.	8-4
GPIO_close	F	Closes previously opened GPIO device.	8-5
GPIO_config	F	Configures GPIO device using configuration structure.	8-6
GPIO_getConfig	F	Reads the current GPIO configuration values.	8-6
GPIO_getDirection	F	Gets input/output direction of the GPIO pins.	8-7
GPIO_getEventId	F	Gets interrupt request event ID for the given GPIO Pin.	8-7
GPIO_getIrqMode	F	Reads current IRQ configuration.	8-8
GPIO_getPinDelta	F	Detects changed pins.	8-8
GPIO_getSetup	F	Returns the set-up parameters for a GPIO pin.	8-9
GPIO_getStatus	F	Gets enable/disable status of the GPIO pins.	8-9
GPIO_open	F	Opens GPIO device.	8-10
GPIO_pinRead	F	Reads data from a single pin.	8-10
GPIO_pinWrite	F	Writes the value to the specified GPIO pin.	8-11
GPIO_read	F	Reads data values from a group of pins.	8-11
GPIO_reset	F	Resets GPIO device that is already opened.	8-12
GPIO_setDirection	F	Sets input/output direction of the GPIO pins.	8-12
GPIO_setIrqMode	F	Sets IRQ triggering mode.	8-13
GPIO_setStatus	F	Enables/disables the GPIO pins.	8-13
GPIO_setup	F	Sets the parameters for a GPIO pin using the GPIO set-up structure.	8-14
GPIO_write	F	Writes data to group of pins.	8-14

Note: C = Constant; F = Function; S = Structure

8.2 API Reference

GPIO_Config — *GPIO configuration structure*

Structure GPIO_Config

Members

Uint32 ior	GPIO input/output register
Uint32 cior	GPIO configuration register
Uint32 irqA	GPIO interrupt request register A
Uint32 irqB	GPIO interrupt request register B
Uint32 ddior	GPIO delta detect register
Uint32 enr	GPIO Mux select register

Description This is the GPIO configuration structure used to set up a GPIO device. User can create and initialize this structure and then pass its address to the GPIO_config function.

GPIO_DEVICE_CNT — *GPIO device count*

Constant GPIO_DEVICE_CNT

GPIO_OPEN_RESET — *GPIO open reset flag*

Constant GPIO_OPEN_RESET

Description This flag is used while opening GPIO device. To open with reset use GPIO_OPEN_RESET otherwise 0.

Example see GPIO_open

GPIO_Setup — *GPIO set-up structure*

Structure GPIO_Setup

Members

Uint16 enab	GPIO pin enable/disable(multiplexing) see GPIO_setStatus
Uint16 dir	GPIO input/output direction see L
Uint16 irqMode	GPIO interrupt trigger mode see L

Description This is the GPIO setup structure used to set up a GPIO pin. User can create and initialize this structure and then pass its address to the GPIO_setup function with pin ID.

GPIO_clearPinDelta — *Clear pin outputs*

Function

```
void  GPIO_clearPinDelta(
    GPIO_Handle       hGpio,
    Uint32            pinMask
)
```

Arguments

hGpio	Device handle
pinMask	GPIO pin mask

Return Value none

Description Used to clear bits of given pins in Delta Detect Register. Available pin IDs are as follows (To get pinMask, user can OR them for grouping pins):

- GPIO_PIN0
- GPIO_PIN1
- GPIO_PIN2
- GPIO_PIN3
- GPIO_PIN4
- GPIO_PIN5
- GPIO_PIN6
- GPIO_PIN7
- GPIO_PIN8
- GPIO_PIN9
- GPIO_PIN10
- GPIO_PIN11
- GPIO_PIN12

- ❑ GPIO_PIN13
- ❑ GPIO_PIN14
- ❑ GPIO_PIN15
- ❑ GPIO_PIN16
- ❑ GPIO_PIN17
- ❑ GPIO_PIN18
- ❑ GPIO_PIN19

Example

```
GPIO_clearPinDelta(hGpio, 0x005FF0);
```

GPIO_close — *Closes previously opened GPIO device*

Function

```
void GPIO_close(
    GPIO_Handle     hGpio
)
```

Arguments

hGpio   Device handle; see GPIO_open

Return Value

none

Description

Closes a previously opened GPIO device; see GPIO_open. The following tasks are performed:

1) The GPIO event is disabled and cleared.

2) The GPIO registers are set to their default values.

Example

```
GPIO_close(hGpio);
```

GPIO_config — *Configures GPIO device using configuration structure*

Function

```
void  GPIO_config(
    GPIO_Handle       hGpio,
    GPIO_Config       *myConfig
)
```

Arguments

hGpio	Device handle; see GPIO_open
myConfig	Pointer to the configuration structure

Return Value

none

Description

Configures GPIO device using the configuration structure. The values of the structure members are written to GPIO registers.

Example

```
Config MyConfig = {
    0x0u,                   // ior
    0x000FFFFFu             // cior
    0x0u,                   // irqA
    0x0u,                   // irqB
    0x000FFFFFu,            // ddior
    0x000FFFFFu             // enr
};
...
GPIO_config(hGpio, &MyConfig);
```

GPIO_getConfig — *Reads the current GPIO configuration values*

Function

```
void  GPIO_getConfig(
    GPIO_Handle       hGpio,
    GPIO_Config       *config
)
```

Arguments

hGpio	Device handle; see GPIO_open
config	Pointer to the source configuration structure

Return Value

none

Description

Gets the current GPIO configuration values

Example

```
GPIO_Config gpioCfg;
GPIO_getConfig(hGpio, &gpioCfg);
```

GPIO_getDirection *Gets input/output direction of the GPIO pins*

Function

```
Uint32  GPIO_getDirection(
    GPIO_Handle     hGpio,
    Uint32          pinMask
)
```

Arguments

hGpio	Device handle; see GPIO_open
pinMask	I/O pin mask

Return Value Uint32

Description Use this function to get the input/output direction of the pins specified by pinMask. See GPIO_clearPinDelta for pinMask specification dir – GPIO_IN, GPIO_OUT. Extract the return value for the corresponding pin ID

Example

```
PinMaskDir = GPIO_getDirection(hGpio, 0x001FFFE0u);
```

GPIO_getEventId *Gets interrupt request event ID for the given GPIO pin*

Function

```
Uint16  GPIO_getEventId(
    GPIO_Handle     hGpio,
    Uint32          ePinId
)
```

Arguments

hGpio	Device handle; see GPIO_open
ePinId	GPIO pin ID

Return Value Uint16   IRQ event ID for the GPIO device

Description Use this function to obtain the event ID for the GPIO device. See GPIO_clearPinDelta for available pin IDs return values:

- IRQ_EVT_GPIO0 – Pin ID 0
- IRQ_EVT_GPIO1 – Pin ID 1
- IRQ_EVT_GPIO2 – Pin ID 2
- IRQ_EVT_GPIO3 – Pin ID 3
- IRQ_EVT_GPIO4 – Pin ID 4 to 19
- IRQ_EVT_KBIO_COL – KBIO row pins (8 to 15)
- IRQ_EVT_KBIO_ROW – KBIO column pins (0 to 7)
- 0xFFFF – ERROR

Example

```
GpioEventID = GPIO_getEventId(hGpio, GPIO_PIN0);
IRQ_enable(GpioEventID);
```

GPIO_getIrqMode *Reads current IRQ configuration*

Function

```
Uint16  GPIO_getIrqMode(
    GPIO_Handle     hGpio,
    Uint32          ePinId
)
```

Arguments

hGpio	device handle
ePinId	GPIO pin ID

Return Value

Uint16	Current IRQ configuration

Description

Use this function to get the IRQ configuration return values:

- 0 – GPIO_IRQ_DIS (Disable IRQ)
- 1 – GPIO_IRQ_RISE (IRQ generated on rising edge)
- 2 – GPIO_IRQ_FALL (IRQ generated on falling edge)
- 3 – GPIO_IRQ_STCH (IRQ generated on state change)

Example

```
IrqMode = GPIO_getIrqMode (hGpio, GPIO_PIN3);
```

GPIO_getPinDelta *Detects changed pins*

Function

```
Uint32  GPIO_getPinDelta(
    GPIO_Handle     hGpio,
    Uint32          pinMask
)
```

Arguments

hGpio	Device handle
pinMask	GPIO pin mask

Return Value

Uint32

Description

Use this function to read the change in the pins specified by pinMask. See GPIO_clearPinDelta for pinMask specification. Extract the return value for the corresponding pin ID.

Example

```
deltaPattern = GPIO_getPinDelta(hGpio, 0x005FF0);
```

GPIO_getSetup

Returns the set-up parameters for a GPIO pin

Function

```
void GPIO_getSetup(
    GPIO_Handle     hGpio,
    Uint32          ePinId,
    GPIO_Setup      *setup
)
```

Arguments

hGpio	Device handle; see GPIO_open
ePinId	GPIO pin ID
setup	Set-up structure

Return Value

none

Description

Returns the set-up values used for a specified pin. Pin mask cannot be used for this API. See GPIO_clearPinDelta for available pin IDs. See GPIO_Setup for setup parameters.

Example

```
GPIO_Setup MySetup;
...
GPIO_getSetup(hGpio, GPIO_PIN0, &MySetup);
```

GPIO_getStatus

Gets enable/disable status of the GPIO pins

Function

```
Uint32 GPIO_getStatus(
    GPIO_Handle     hGpio,
    Uint32          pinMask
)
```

Arguments

hGpio	Device handle; see option
pinMask	I/O pin mask

Return Value

Uint32

Description

Use this function to get the enable/disable status of the GPIO pins specified by pinMask. See GPIO_clearPinDelta for pinMask specification modes.

- GPIO_ENABLE – GPIO enable
- GPIO_DISABLE – GPIO disable,configuration for other I/O signal

Extract the return value for the corresponding pin ID.

Example

```
PinStatus = GPIO_getStatus(hGpio, pinMask);
```

GPIO_open — *Opens GPIO device*

Function

```
GPIO_Handle  GPIO_open(
    Uint16        devNum,
    Uint16        flags
)
```

Arguments

devNum — Specifies the device to be opened:
- GPIO_DEV0
- GPIO_DEV1
- GPIO_DEV_ANY

flags — Open flags
GPIO_OPEN_RESET – resets the GPIO

Return Value

GPIO_Handle — Device handle
INV – open failed

Description

Before a GPIO can be used, it must be ‘opened’ using this function. Once opened it cannot be opened again until it is ‘closed’ (see GPIO_close). The return value is a unique device handle that is used in subsequent GPIO API calls. If the open fails, ‘INV’ is returned.

If the GPIO_OPEN_RESET flag is specified, the GPIO device registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example

```
GPIO_Handle hGpio;
...
hGpio = GPIO_open(GPIO_DEV0, GPIO_OPEN_RESET);
```

GPIO_pinRead — *Reads data from a single pin*

Function

```
Uint16  GPIO_pinRead(
    GPIO_Handle      hGpio,
    Uint32        ePinId
)
```

Arguments

hGpio — Device handle

ePinId — Pin ID

Return Value

Uint16

Description

Use this function to read data from the given pin. See GPIO_clearPinDelta for available pin IDs.

Example

```
pinVal= GPIO_pinRead(hGpio, GPIO_PIN8);
```

GPIO_pinWrite *Writes the value to the specified GPIO pin*

Function

```
void  GPIO_pinWrite(
    GPIO_Handle       hGpio,
    Uint32            ePinId,
    Uint32            val
)
```

Arguments

hGpio	Device handle
ePinId	GPIO pin ID
val	bit value

Return Value none

Description Use this function to write the value to GPIO pin. See GPIO_clearPinDelta for available pin IDs.

Example

```
GPIO_pinWrite(hGpio,GPIO_PIN2,1);
```

GPIO_read *Reads data values from a group of pins*

Function

```
Uint32  GPIO_read(
    GPIO_Handle       hGpio,
    Uint32            pinMask
)
```

Arguments

hGpio	Device handle
pinMask	pin mask

Return Value Uint32

Description Use this function to read the pin values specified by pinMask. See GPIO_clearPinDelta for pinMask specification. Extract the return value for the corresponding pin ID.

Example

```
pinVal = GPIO_read(hGpio, GPIO_PIN2 | GPIO_PIN3 | GPIO_PIN6 | GPIO_PIN10);
pinVal = GPIO_read(hGpio, 0x005FF0);
```

GPIO_reset — *Resets GPIO device that is already opened*

Function

```
void GPIO_reset(
    GPIO_Handle     hGpio
)
```

Arguments

hGpio	Device handle; see GPIO_open

Return Value none

Description Disables and clears the interrupt event and sets the GPIO registers to their default values.

Example

```
GPIO_reset(hGpio);
```

GPIO_setDirection — *Sets input/output direction of the GPIO pins*

Function

```
void GPIO_setDirection(
    GPIO_Handle     hGpio,
    Uint32          pinMask,
    Uint16          dir
)
```

Arguments

hGpio	Device handle; see GPIO_open
pinMask	I/O pin mask
dir	I/O direction

Return Value none

Description Use this function to set the input/output direction of the pins specified by pinMask. See GPIO_clearPinDelta for pinMask specification dir – GPIO_IN, GPIO_OUT.

Example

```
GPIO_setDirection(hGpio, 0x001FFFE0u, GPIO_IN);
```

GPIO_setIrqMode *Sets IRQ triggering mode*

Function

```
void  GPIO_setIrqMode(
    GPIO_Handle     hGpio,
    Uint32          pinMask,
    Uint16          eMode
)
```

Arguments

hGpio	Device handle
pinMask	GPIO pin mask
eMode	IRQ modes (enumerated)

Return Value none

Description Enables/disables the interrupts in the pins specified by the pinMask to rising edge, falling edge, or on-state change. See GPIO_getIrqMode for available IRQ Modes.

Example

```
GPIO_setIrqMode (hGpio, GPIO_PIN3 | GPIO_PIN5 | GPIO_PIN7, GPIO_IRQ_FALL);
```

GPIO_setStatus *Enables/disables the GPIO pins*

Function

```
void  GPIO_setStatus(
    GPIO_Handle     hGpio,
    Uint32          pinMask,
    Uint16          mode
)
```

Arguments

hGpio	Device handle; see option
pinMask	I/O pin mask
mode	Enable/disable mode

Return Value none

Description Use this function to set the enable/disable status of the GPIO pins specified by pinMask. See GPIO_clearPinDelta for pinMask specification modes.

- GPIO_ENABLE – GPIO enable
- GPIO_DISABLE – GPIO disable, configuration for other I/O signal

Example

```
PinStatus = GPIO_setStatus(hGpio, pinMask, GPIO_ENABLE);
```

GPIO_setup
Sets the parameters for a GPIO pin using the GPIO set-up structure

Function

```
void GPIO_setup(
    GPIO_Handle     hGpio,
    Uint32          ePinId,
    GPIO_Setup      *setup
)
```

Arguments

hGpio	Device handle
ePinId	GPIO pin ID
setup	Initialization structure

Return Value none

Description This function sets up the parameters for a GPIO pin using the GPIO_Setup structure. See GPIO_clearPinDelta for available pin IDs.

Example

```
GPIO_Setup MySetup = {
    1,
    1,
    GPIO_IRQ_RISE
};
...
GPIO_setup(hGpio, GPIO_PIN0, &MySetup);
```

GPIO_write
Writes data to group of pins

Function

```
void GPIO_write(
    GPIO_Handle     hGpio,
    Uint32          pinMask,
    Uint32          bitPattern
)
```

Arguments

hGpio	Device handle
pinMask	GPIO pin mask
bitPattern	bit value pattern

Return Value none

Description Use this function to write the values given as bitPattern to GPIO pins specified by pinMask. See GPIO_clearPinDelta for pinMask specification.

Example

```
GPIO_write(hGpio, GPIO_PIN2 | GPIO_PIN3 | GPIO_PIN6 | GPIO_PIN10, 0x80C);
GPIO_write(hGpio, 0x005FF0, 0x80C);
```

8.3 Register and Field Names

Table 8–2. GPIO Module Register and Field Names

Register Name	Field Name(s)
GPIO_IOR	IO
GPIO_CIOR	CIO
GPIO_IRQA	IRQA
GPIO_IRQB	IRQB
GPIO_DDIOR	DDCT
GPIO_ENR	ENAB

Note: All fields are Read/Write

Chapter 9

I2C Module

This module provides an interface between between VC547x system interface bus and I2C bus. The VC547x system bus through the master I2C interface module can control the external peripheral devices on the I2C bus.

Topic **Page**

9.1 Overview

Table 9–1. I2C Descriptions

Syntax	Type	Description	Page
I2C_Config	S	I2C configuration structure.	9-4
I2C_DEVICE_CNT	C	I2C device count.	9-4
I2C_OPEN_RESET	C	I2C reset flag, used while opening.	9-4
I2C_Setup	S	I2C set-up structure.	9-5
I2C_chkFifoEmpty	F	Gets empty/not empty status of FIFO.	9-5
I2C_chkFifoFull	F	Checks full/not full status of FIFO.	9-6
I2C_close	F	Closes previously opened I2C device.	9-6
I2C_config	F	Configures I2C using configuration structure.	9-7
I2C_eventDisable	F	Disables the interrupt event.	9-7
I2C_eventEnable	F	Enables the interrupt event.	9-7
I2C_fget	F	Reads one byte data from the slave.	9-8
I2C_fput	F	Sends one byte data to the slave.	9-8
I2C_getConfig	F	Reads the current I2C configuration values.	9-9
I2C_getSetup	F	Gets the current I2C set up.	9-9
I2C_getSlaveAddr	F	Gets the slave device ID.	9-10
I2C_getSlaveSubAddr	F	Gets the slave device internal address.	9-10
I2C_isFree	F	Checks free/not free status of FIFO.	9-10
I2C_open	F	Opens I2C device for use.	9-11
I2C_read	F	Performs an I2C read.	9-11
I2C_reset	F	Resets the I2C device.	9-12
I2C_resetFifo	F	Resets the FIFO.	9-12
I2C_setSlaveAddr	F	Sets the slave address for data transfer.	9-13
I2C_setSlaveSubAddr	F	Sets the slave sub-address for data transfer.	9-13
I2C_setTxRate	F	Sets the data transmission rate.	9-14

Note: C = Constant; F = Function; S = Structure

Table 9–1. I2C Descriptions (Continued)

Syntax	Type	Description	Page
I2C_setup	F	Sets up and initiates I2C operation.	9-15
I2C_write	F	Performs an I2C write.	9-15

Note: C = Constant; F = Function; S = Structure

9.2 API Reference

I2C_Config *I2C configuration structure*

Structure I2C_Config

Members

Uint32 devr	Device slave address register
Uint32 addr	Device slave subaddress register
Uint32 dwr	Data write register
Uint32 drr	Data read register
Uint32 cmdr	Command register
Uint32 cfr	Configuration FIFO register
Uint32 ccr	Configuration clock register
Uint32 ccfr	Configuration clock functional reference register
Uint32 sfr	Status FIFO register (read only)
Uint32 sar	Status activity register

Description This is the I2C configuration structure used to configure (register-based) I2C device. You create and initialize this structure and then pass its address to the I2C_config function.

I2C_DEVICE_CNT *I2C device count*

Constant I2C_DEVICE_CNT

I2C_OPEN_RESET *I2C reset flag*

Constant I2C_OPEN_RESET

Description This flag is used while opening the I2C device. To open with reset; use I2C_OPEN_RESET otherwise use 0.

Example See I2C_open

I2C_Setup — *I2C set-up structure*

Structure I2C_Setup

Members

Uint16 devAddr	device identification code for I2C bus slave device
Uint16 devIntAddr	I2C slave device internal register address
Uint16 irqSet	Interrupt request enable/disable
Uint16 combRead	Simple or combined read
Uint16 rwAccess	I2C bus read/write access
Uint16 fifoSize	Size of the FIFO to generate FIFO_FULL
Uint16 spkFactor	Spike filter factor
Uint16 preClkDiv1	Prescale clock divide factor
Uint16 clkRefDiv2	Functional clock reference

Description This structure is used to set up and initiate the I2C operation. You create and initialize this structure and then pass its address to the I2C_setup function.

I2C_chkFifoEmpty — *Gets empty/not empty status of FIFO*

Function

```
Bool  I2C_chkFifoEmpty(
    I2C_Handle       hI2c
)
```

Arguments hI2c — Device handle; see I2C_open

Return Value Bool
- 1 – empty
- 0 – not empty

Description Gets the FIFO empty/not empty status.

Example

```
Bool status = I2C_chkFifoEmpty(hI2c);
```

I2C_chkFifoFull — *Checks full/not full status of FIFO*

Function

```
Bool  I2C_chkFifoFull(
    I2C_Handle        hI2c
)
```

Arguments

hI2c   Device handle; see I2C_open

Return Value

Bool   1 – full
        0 – not full

Description

Gets the FIFO full/not full status.

Example

```
Bool status = I2C_chkFifoFull(hI2c);
```

I2C_close — *Closes previously opened I2C device*

Function

```
void  I2C_close(
    I2C_Handle        hI2c
)
```

Arguments

hI2c   Device handle; see I2C_open

Return Value

none

Description

Closes a previously opened I2C device (see I2C_open). The following tasks are performed:

- The I2C event is disabled and cleared.
- The I2C registers are set to their default values.

Example

```
I2C_close(hClkm);
```

I2C_config — *Configures I2C using configuration structure*

Function
```
void I2C_config(
    I2C_Handle      hClkm,
    I2C_Config      *myConfig
)
```

Arguments

hClkm — Device handle; see I2C_open

myConfig — Pointer to the configuration structure

Return Value none

Description Configures the I2C device using the configuration structure. The values of the structure variables are written to the I2C registers.

Example
```
Config MyConfig
...
I2C_config(hI2c, &MyConfig);
```

I2C_eventDisable — *Disables the interrupt event*

Function
```
void I2C_eventDisable(
    I2C_Handle      hI2c
)
```

Arguments hI2c — Device handle; see I2C_open

Return Value none

Description Disables the interrupt event.

Example
```
I2C_Handle hI2c;
.....
I2C_eventDisable(hI2c);
```

I2C_eventEnable — *Enables the interrupt event*

Function
```
void I2C_eventEnable(
    2C_Handle      hI2c
)
```

Arguments hI2c — Device handle; see I2C_open

Return Value none

Description Enables the interrupt event.

Example
```
I2C_Handle hI2c;
.....
I2C_eventEnable(hI2c);
```

I2C_fget — *Reads one byte data from the slave*

Function

```
Uint16  I2C_fget(
    I2C_Handle      hI2c,
    Int8            *data
)
```

Arguments

hI2c   Device handle; see I2C_open

data   read data pointer

Return Value   Uint16

Description   This function can be used to read one byte data from the slave. Return values:

- 0 – (I2C_NO_ERROR)
- 1 – (I2C_ERROR_DEVICE)
- 2 – (I2C_ERROR_DATA)

Example

```
I2C_Handle  hI2c;
Int8 data
.......
I2C_fget(hI2c,&data);
```

I2C_fput — *Sends one byte data to the slave*

Function

```
Uint16  I2C_fput(
    I2C_Handle      hI2c,
    Int8            data
)
```

Arguments

hI2c   Device handle; see I2C_open

data   8-bit data to be sent to slave device

Return Value   Uint16

Description   This function can be used to send one byte data to the slave. The function returns the status of transmission/reception. Return values:

- 0 – (I2C_NO_ERROR)
- 1 – (I2C_ERROR_DEVICE)
- 2 – (I2C_ERROR_DATA)

Example

```
I2C_Handle  hI2c;
.......
I2C_fput(hI2c,0xff);
```

I2C_getConfig

Gets the current I2C configuration values

Function

```
void  I2C_getConfig(
     I2C_Handle        hI2C,
     I2C_Config        *myConfig
)
```

Arguments

hI2C	Device handle; see I2C_open
myConfig	Pointer to the configuration structure

Return Value none

Description Gets the current I2C configuration values.

Example

```
I2C_Config i2cCfg;
getConfig(hI2c, &i2cCfg);
```

I2C_getSetup

Gets the current I2C set up

Function

```
void  I2C_getSetup(
     I2C_Handle        hI2c,
     I2C_Setup         *mySetup
)
```

Arguments

hI2c	Device handle; see I2C_open
mySetup	Set-up structure

Return Value none

Description Gets the current I2C set up.

Example

```
I2C_Handle hI2c;
I2C_Setup curSetup;
....
I2C_getSetup(hI2c,&curSetup);
```

I2C_getSlaveAddr *Gets the slave device ID*

Function
```
Uint16  I2C_getSlaveAddr(
    I2C_Handle       hI2c
)
```

Arguments hI2c Device handle; see I2C_open

Return Value Uint16

Description Gets the slave device identification code specified for data transfer.

Example
```
I2C_Handle hI2c;
Uint16 devId;
.....
devId = getSlaveAddr(hI2c);
```

I2C_getSlaveSubAddr *Gets the slave device internal address*

Function
```
Uint16  I2C_getSlaveSubAddr(
    I2C_Handle       hI2c
)
```

Arguments hI2c Device handle; see I2C_open

Return Value Uint16

Description Gets the slave device internal register address.

Example
```
I2C_Handle hI2c;
Uint16 devId;
.....
devId = getSlaveSubAddr(hI2c);
```

I2C_isFree *Checks free/not free status of FIFO*

Function
```
Bool  I2C_isFree(
    I2C_Handle       hI2c
)
```

Arguments hI2c Device handle; see I2C_open

Return Value Bool
1 – free
0 – not free

Description Gets the FIFO free/not free status. Free Indicates the I2C bus transfer is completed.

Example
```
Bool status = I2C_isFree(hI2c);
```

I2C_open — *Opens I2C device for use*

Function

```
I2C_Handle  I2C_open(
    Uint16        devNum,
    Uint16        flags
)
```

Arguments

devNum	specifies the device to be opened
flags	Open flags I2C_OPEN_RESET – resets the I2C

Return Value

I2C_Handle	Device handle INV – open failed

Description

Before the I2C device can be used, it must be 'opened' using this function. Once opened it cannot be opened again until it is 'closed' (see I2C_close). The return value is a unique device handle that is used in subsequent I2C API calls. If the open fails, 'INV' is returned. If the I2C_OPEN_RESET flag is specified, the I2C module registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example

```
Handle hI2c;
...
hI2c = I2C_open(I2C_DEV0, I2C_OPEN_RESET);
```

I2C_read — *Performs an I2C read*

Function

```
Uint16  I2C_read(
    I2C_Handle      hI2c,
    Int8          *data,
    Uint16        dataLen
)
```

Arguments

hI2c	Device handle; see I2C_open
data	read data pointer
dataLen	read data count

Return Value

Uint16

Description

This function can be used to read dataLen number of bytes from the slave. Note that due to bugs in the I2C device in the Orion chip (VC547x), the I2C_read function has the following limitations:

1) As the device does not have a read FIFO, it is incapable of accepting more than one byte during any given read operation.

2) Since the VC547x sends an ACK instead of an NAK after the completion of a read operation, slave devices that require an NAK to appropriately complete a read operation may hang. The user has to ensure that such slave devices are brought out of their hung-up state after a read operation. Return values:

   - 0 – (I2C_NO_ERROR)
   - 1 – (I2C_ERROR_DEVICE)
   - 2 – (I2C_ERROR_DATA)

Example

```
I2C_Handle  hI2c;
Int8 data[16];
.......
I2C_read(hI2c,data,10);
```

I2C_reset — *Resets the I2C device*

Function

void I2C_reset(
    I2C_Handle   hI2c
)

Arguments hI2c   Device handle; see I2C_open

Return Value none

Description Sets the I2C registers to their default values.

Example

```
I2C_Handle hI2c;
......
I2C_reset(hI2c);
```

I2C_resetFifo — *Resets the FIFO*

Function

void I2C_resetFifo(
    I2C_Handle   hI2c
)

Arguments hI2c   Device handle; see I2C_open

Return Value none

Description Resets the FIFO. FIFO has to be reset before each data transfer.

Example

```
I2C_Handle hI2c;
....
I2C_resetFifo(hI2c);
```

I2C_setSlaveAddr *Sets the slave address for data transfer*

Function

```
void  I2C_setSlaveAddr(
    I2C_Handle      hI2c,
    Uint16          devAddr,
    Uint16          subAddr
)
```

Arguments

hI2c	Device handle; see I2C_open
devAddr	Slave device address
subAddr	Slave device internal subaddress

Return Value none

Description Sets the slave device and subaddress for data transfer.

Example

```
I2C_Handle hI2c;
......
setSlaveAddr(hI2c,0x50,0xA5);
```

I2C_setSlaveSubAddr *Sets the slave subaddress for data transfer*

Function

```
void  I2C_setSlaveSubAddr(
    2C_Handle      hI2c,
    Uint16         subAddr
)
```

Arguments

hI2c	Device handle; see I2C_open
subAddr	Slave device internal subaddress

Return Value none

Description Sets the slave subaddress for data transfer.

Example

```
I2C_Handle hI2c;
......
setSlaveSubAddr(hI2c,0xA5);
```

I2C_setTxRate *Sets the data transmission rate*

Function

```
void  I2C_setTxRate(
    I2C_Handle      hI2c,
    Uint16          div1,
    Uint16          div2
)
```

Arguments

hI2c	Device handle; see I2C_open
div1	Prescale clock divider factor
div2	Functional clock divider factor

Return Value none

Description Sets the transmission rate by using prescale clock divisor factor (div1) and functional clock divisor factor (div2).

CLK_FUNC_REF = I2C_clk/(Div1*[Div2+1])
SCL_OUT = CLK_FUNC_REF/2

Possible div1 values are:

- I2C_PTV_DIV2 – 2
- I2C_PTV_DIV4 – 4
- I2C_PTV_DIV8 – 8
- I2C_PTV_DIV16 – 16
- I2C_PTV_DIV32 – 32
- I2C_PTV_DIV64 – 64
- I2C_PTV_DIV128 – 128
- I2C_PTV_DIV256 – 256

Div2 can be any values ranging from 1 to 127.

Example

```
I2C_Handle hI2c;
...
I2C_setTxRate(hI2c,I2C_PTV_DIV16,27);
```

I2C_setup

Sets up and initiates I2C operation

Function

```
void  I2C_setup(
    I2C_Handle      hI2c,
    I2C_Setup       *mySetup
)
```

Arguments

hI2c	Device handle; see I2C_open
mySetup	Initialized set-up structure

Return Value none

Description Sets up and initiates the I2C operation using the set-up structure.

Example

```
I2C_Setup mySetup;
.....
I2C_setup(hI2c,&mySetup);
```

I2C_write

Performs an I2C write

Function

```
Uint16  I2C_write(
    I2C_Handle      hI2c,
    Int8            *data,
    Uint16          dataLen
)
```

Arguments

hI2c	Device handle; see I2C_open
data	8-bit data to be sent to slave device
dataLen	Number of data to be sent

Return Value Uint16

Description Performs master/slave transmission for specified number of data. Return values:

- 0 – (I2C_NO_ERROR)
- 1 – (I2C_ERROR_DEVICE)
- 2 – (I2C_ERROR_DATA)

Example

```
I2C_Handle  hI2c;
Int8 data[10] ={1,2,3,4,5,6,7,8,9,10};
.......
I2C_write(hI2c,data,10);
```

9.3 Register and Field Names

Table 9–2. I2C Module Register and Field Names

Register Name	Field Name(s)
I2C_DEVR	DEVICE
I2C_ADDR	ADDRESS
I2C_DWR	DATAWRITE
I2C_DRR	DATAREAD
I2C_CMDR	IRQMASK, COMBREAD, RNW, START, SOFTRESET
I2C_CFR	FIFOSIZE
I2C_CCR	SPIKEFAC, PTV
I2C_CCFR	CLKREF
I2C_SFR	READCPT (R), FIFOEMPTY (R), FIFOFULL (R)
I2C_SAR	INTR (R), IDLE (R), ERRORDEVICE (R), ERRORDATA (R)

Note: R = Read only; fields not marked are Read/Write

Chapter 10

IRQ Module

The IRQ module provides APIs for interfacing to the on-chip interrupt handler. The ARM MCU interrupt handler prioritizes and masks interrupts from up to 16 interrupt sources. It also provides for individually routing each interrupt source to one of the two interrupt lines, IRQ and FIQ, of the ARM MCU.

Topic **Page**

10.1 Overview

Table 10–1. IRQ Descriptions

Syntax	Type	Description	Page
IRQ_EVENT_CNT	C	Event count.	10-3
IRQ_INT_CNT	C	Interrupt count.	10-3
IRQ_Setup	S	IRQ set-up structure.	10-3
IRQ_clear	F	Clears an interrupt that has been latched.	10-4
IRQ_disable	F	Disables a specific interrupt.	10-4
IRQ_enable	F	Enables a specific interrupt.	10-4
IRQ_getPriority	F	Gets priority of an event.	10-5
IRQ_getRoute	F	Gets current route of an event.	10-5
IRQ_getSense	F	Gets the edge sense setting for an event.	10-5
IRQ_getSetup	F	Returns the set-up structure for an interrupt.	10-6
IRQ_globalDisable	F	Disables IRQ/FIQ exceptions.	10-6
IRQ_globalEnable	F	Enables IRQ/FIQ exceptions.	10-6
IRQ_globalRestore	F	Restores IRQ/FIQ exceptions.	10-7
IRQ_initDispatcher	F	Sets up top-level IRQ and FIQ dispatchers.	10-7
IRQ_plug	F	Attaches an Interrupt Service Routine to an interrupt.	10-8
IRQ_reset	F	Clears and disables an interrupt.	10-8
IRQ_restore	F	Restores a specific interrupt.	10-9
IRQ_setPriority	F	Sets priority of an event.	10-9
IRQ_setRoute	F	Sets current route of an event.	10-10
IRQ_setSense	F	Alters the sense edge setting.	10-10
IRQ_setup	F	Configures an interrupt using set-up structure.	10-11
IRQ_test	F	Checks for latched interrupt.	10-11

Note: C = Constant; F = Function; S = Structure

10.2 API Reference

IRQ_EVENT_CNT *Event count*

Constant IRQ_EVENT_CNT

IRQ_INT_CNT *Interrupt count*

Constant IRQ_INT_CNT

IRQ_Setup *IRQ set-up structure*

Structure IRQ_Setup

Members

IRQ_IsrPtr isrAddr	Pointer to the event's Interrupt Service Routine (ISR)
Uint16 priority	The interrupt priority IRQ_PRIORITY_DEFAULT
Uint16 route	Routing: ❑ IRQ_ROUTE_FIQ ❑ IRQ_ROUTE_IRQ
Uint16 sense	Sense edge: ❑ IRQ_SENSE_LOWLEVEL – trigger on low level ❑ IRQ_SENSE_FALLINGEDGE – trigger on falling edge

Description This is the IRQ set-up structure. The set-up structure can be used to configure an interrupt (using IRQ_setup) or to get the current setup parameters (using IRQ_getSetup). The set-up structure contains members representing the address of the Interrupt Service Routine (ISR), the interrupt priority, the interrupt routing (whether it is routed to the IRQ or FIQ pin of the ARM core) and the interrupt sense setting (that specifies if the interrupt is falling-edge or low-level sensitive).

IRQ_clear — *Clears an interrupt that has been latched*

Function

void IRQ_clear(
    Uint16 eventId
)

Arguments

eventId   Event ID of the interrupt

Return Value

none

Description

Clears an interrupt that has been latched. Clearing an interrupt (from within another handler) prevents the interrupt handler from queuing that interrupt after the current one has been serviced.

Example

```
IRQ_clear (IRQ_TIMER_TINT0);
```

IRQ_disable — *Disables a specific interrupt*

Function

Uint32 IRQ_disable(
    Uint16 eventId
)

Arguments

eventId   Event ID of the interrupt

Return Value

Uint32   The old interrupt state

Description

Disables (mask) the interrupt with a particular event ID. Returns the previous status (enabled/disabled) of the interrupt.

Example

```
oldState = IRQ_disable(IRQ_TIMER_TINT0);
```

IRQ_enable — *Enables a specific interrupt*

Function

void IRQ_enable(
    Uint16 eventId
)

Arguments

eventId   Event ID of the interrupt

Return Value

none

Description

Enables (unmask) the interrupt with a particular event ID.

Example

```
IRQ_enable(IRQ_TIMER_TINT0);
```

IRQ_getPriority — *Gets priority of an event*

Function

```
Uint16  IRQ_getPriority(
    Uint16          eventId
)
```

Arguments eventId — Event ID of the interrupt

Return Value Uint16 — The interrupt priority (0 to 15)

Description Returns the interrupt priority of the event.

Example

```
Uint16 pri = IRQ_getPriority (IRQ_TIMER_TINT0);
```

IRQ_getRoute — *Gets current route of an event*

Function

```
Uint16  IRQ_getRoute(
    Uint16          eventId
)
```

Arguments eventId — Event ID of the interrupt

Return Value Uint16 — Route:

- IRQ_ROUTE_FIQ
- IRQ_ROUTE_IRQ

Description Returns a value indicating whether the interrupt associated with the event is being routed to IRQ or FIQ.

Example

```
Uint16 route = IRQ_getRoute (IRQ_TIMER_TINT0);
```

IRQ_getSense — *Gets the edge sense setting for an event*

Function

```
Uint16  IRQ_getSense(
    Uint16          eventId
)
```

Arguments eventId — Event ID

Return Value Uint16 — The current sense edge setting

Description Return a constant indicating the sense edge setting for an event.

Example

```
Uint16 sensEdge = IRQ_getSense(IRQ_TIMER_TINT0);
```

IRQ_getSetup *Returns the set-up structure for an interrupt*

Function

```
void  IRQ_getSetup(
    Uint16          eventId,
    IRQ_Setup       *setup
)
```

Arguments

eventId	Event ID of the interrupt
setup	Set-up structure

Return Value none

Description Returns the set-up structure associated with a particular interrupt.

Example

```
IRQ_Setup mySetup;
IRQ_getSetup(IRQ_TIMER_TINT0, &mySetup);
```

IRQ_globalDisable *Disables IRQ/FIQ exceptions*

Function

```
Uint32  IRQ_globalDisable(
    void
)
```

Arguments none

Return Value Uint32

Description Disables both the IRQ and FIQ exceptions. The function returns the previous status for both in a mask that can be used while calling IRQ_globalRestore().

Example

```
Uint32 oldGie = IRQ_globalDisable();
```

IRQ_globalEnable *Enables IRQ/FIQ exceptions*

Function

```
void  IRQ_globalEnable(
    void
)
```

Arguments none

Return Value none

Description Enables both the IRQ and FIQ exceptions.

Example

```
IRQ_globalEnable();
```

IRQ_globalRestore *Restores IRQ/FIQ exceptions*

Function

```
void IRQ_globalRestore(
    Uint32      gie
)
```

Arguments gie Restore mask

Return Value none

Description Restores the exception enable mask. This function will usually be used in conjunction with IRQ_globalDisable to demarcate un-interruptible sections of application code.

Example

```
Uint32 oldGie = IRQ_globalDisable();
...
// critical code section
...
IRQ_globalRestore(oldGie);
```

IRQ_initDispatcher *Sets up top-level IRQ and FIQ dispatchers*

Function

```
void IRQ_initDispatcher(
    void
)
```

Arguments none

Return Value none

Description Plugs in the internal IRQ and FIQ dispatcher and initializes CSL internal tables for dispatch. This function should be called if the user wants the CSL to use its internal dispatcher. Hence, it *must* be called before using IRQ_setup, IRQ_setupArgs, IRQ_plug, etc.

Example

```
IRQ_initDispatcher ( );
```

IRQ_plug — *Attaches an Interrupt Service Routine to an interrupt*

Function

```
IRQ_IsrPtr  IRQ_plug(
    Uint16          eventId,
    IRQ_IsrPtr      isrAddr
)
```

Arguments

eventId	Event ID of the interrupt
isrAddr	The ISR's address

Return Value

IRQ_IsrPtr   Address of the previous ISR

Description

Plugs an Interrupt Service Routine (ISR) to an interrupt. The function returns the address of the previously hooked ISR.

Example

```
IRQ_IsrPtr oldIsr;
oldIsr = IRQ_plug(IRQ_TIMER_TINT0, newIsrFunc);
```

IRQ_reset — *Clears and disables an interrupt*

Function

```
void  IRQ_reset(
    Uint16          eventId
)
```

Arguments

eventId   Event ID of the interrupt

Return Value

none

Description

Clears and disables the interrupt associated with a particular event ID. Clearing the interrupt ensures that its ISR is not invoked (because of a currently latched event) when it is re-enabled at a later time.

Example

```
IRQ_reset (IRQ_TIMER_TINT0);
```

IRQ_restore

Restores a specific interrupt

Function

```
void IRQ_restore(
    Uint16        eventId,
    Uint32        ieState
)
```

Arguments

eventId	Event ID of the interrupt
ieState	The interrupt state to restore

Return Value

none

Description

Restores the status (enabled/disabled) of the interrupt associated with the specified event.

Example

```
Uint32 stat = IRQ_disable(IRQ_TIMER_TINT0);
...
IRQ_restore(IRQ_TIMER_TINT0, stat);
```

IRQ_setPriority

Sets priority of an event

Function

```
void IRQ_setPriority(
    Uint16        eventId,
    Uint16        priority
)
```

Arguments

eventId	Event ID of the interrupt
priority	The interrupt priority (0 to 15)

Return Value

none

Description

Sets the interrupt priority of the event.

Example

```
IRQ_setPriority (IRQ_TIMER_TINT0, 15);
```

IRQ_setRoute *Sets current route of an event*

Function

```
void IRQ_setRoute(
    Uint16        eventId,
    Uint16        route_IRQ_or_FIQ
)
```

Arguments

eventId	Event ID of the interrupt
route_IRQ_or_FIQ	Route to IRQ or FIQ: ❑ IRQ_ROUTE_FIQ ❑ IRQ_ROUTE_IRQ

Return Value none

Description Routes the interrupt associated with the event to IRQ or FIQ.

Example

```
route = IRQ_getRoute (IRQ_TIMER_TINT0);
```

IRQ_setSense *Alters the sense edge setting*

Function

```
void IRQ_setSense(
    Uint16        eventId,
    Uint16        sense
)
```

Arguments

eventId	Event ID
sense	Sense edge: ❑ IRQ_SENSE_LOWLEVEL ❑ IRQ_SENSEFALLINGEDGE

Return Value none

Description Alters the sense edge setting for a particular event.

Example

```
IRQ_getSense(IRQ_TIMER_TINT0, IRQ_SENSE_LOWLEVEL);
```

IRQ_setup *Configures an interrupt using set-up structure*

Function

```
void IRQ_setup(
    Uint16        eventId,
    IRQ_Setup     *setup
)
```

Arguments

eventId Event ID associated with the interrupt

setup Set-up structure

Return Value none

Description Sets up the interrupt handler for a particular interrupt taking in parameters from the provided set-up structure.

Example

```
IRQ_Setup mySetup = {
  &isr_timer,
  IRQ_PRIORITY_DEFAULT,
  IRQ_ROUTE_IRQ,
  IRQ_SENSE_FALLINGEDGE
};
IRQ_setup(IRQ_EVT_TINT0, &mySetup);
```

IRQ_test *Checks for latched interrupt*

Function

```
Bool IRQ_test(
    Uint16        eventId
)
```

Arguments

eventId Event ID of the interrupt

Return Value Bool

Description Checks if a particular interrupt has been latched. Returns TRUE/FALSE.

Example

```
Bool isLatched = IRQ_test(IRQ_TIMER_TINT0);
```

10.3 Register and Field Names

Table 10–2. IRQ Module Register and Field Names

Register Name	Field Name(s)
IRQ_ITR	IRQ
IRQ_MIR	IRQ
IRQ_SIR	IRQ
IIRQ_SFR	IRQ
IRQ_ICR	NEW_FIQ_AGR, NEW_IRQ_AGR
IRQ_ISR	IRQ_SLEEP
IRQ_ILR0	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR1	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR2	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR3	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR4	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR5	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR6	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR7	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR8	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR9	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR10	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR11	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR12	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR13	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR14	SENSE_EDGE, PRIORITY, FIQ
IRQ_ILR15	SENSE_EDGE, PRIORITY, FIQ

Note: All fields are Read/Write

Chapter 11

IRUART Module

The IRUART module is the key component in serial communications subsystem. This module can be used in SIR mode for Infrared communications or in UART mode. This module abstracts the register descriptions provided by the IRUART subsystem and provides APIs that can be used to send/receive data and to configure the IRUART.

Topic **Page**

11.1 Overview

Table 11–1. IRUART Descriptions

Syntax	Type	Description	Page
IRUART_Config	S	IRUART configuration structure.	11-4
IRUART_DEVICE_CNT	C	Number of IRUART devices.	11-6
IRUART_DISABLE	C	Disable flag.	11-6
IRUART_ENABLE	C	Enable flag.	11-6
IRUART_OPEN_RESET	C	Flag used to reset an IRUART device while getting a handle.	11-6
IRUART_Setup	S	IRUART set-up structure.	11-6
IRUART_changeMode	F	Changes the mode of the IRUART device.	11-10
IRUART_close	F	Closes a IRUART device.	11-10
IRUART_config	F	Sets IRUART configuration parameters.	11-11
IRUART_eventDisable	F	Disables IRUART interrupts.	11-12
IRUART_eventEnable	F	Enables IRUART interrupts.	11-13
IRUART_fget	F	Reads a character.	11-14
IRUART_fgets	F	Read a string of characters.	11-14
IRUART_fput	F	Writes a character.	11-15
IRUART_fputs	F	Writes a string of characters.	11-15
IRUART_getBaudRate	F	Gets the baud rate.	11-16
IRUART_getConfig	F	Gets the IRUART configuration parameters.	11-16
IRUART_getEventId	F	Gets the IRQ event of IRUART device.	11-16
IRUART_getFrameStatus	F	Gets the frame status.	11-17
IRUART_getIntType	F	Gets the type of interrupt occurred.	11-17
IRUART_getSetup	F	Gets the initial setup values.	11-17
IRUART_loopBack	F	Enables or disables the IRUART device in loopback mode.	11-18
IRUART_open	F	Opens a IRUART device.	11-18
IRUART_read	F	Reads a buffer of characters.	11-19
IRUART_reset	F	Resets the IRUART device.	11-20

Note: C = Constant; F = Function; S = Structure

Table 11–1. IRUART Descriptions (Continued)

Syntax	Type	Description	Page
IRUART_setBaudRate	F	Sets the baud rate.	11-20
IRUART_setup	F	Sets up initial parameters.	11-21
IRUART_write	F	Writes a buffer of characters.	11-22

Note: C = Constant; F = Function; S = Structure

11.2 API Reference

IRUART_Config *IRUART configuration structure*

Structure IRUART_Config

Members

Uint32 fcr	FIFO control register
Uint32 scr	Status control register
Uint32 lcr	Line control register
Uint32 lsr	Line status register
Uint32 ssr	Supplementary status register
Uint32 mcr	Modem control register
Uint32 msr	Modem status register
Uint32 ier	Interrupt enable register
Uint32 isr	Interrupt status register
Uint32 efr	Enhanced feature register
Uint32 xon1	XON1 character register
Uint32 xon2	XON2 character register
Uint32 xoff1	XOFF1 character register
Uint32 xoff2	XOFF2 character register
Uint32 spr	Scratch-pad register
Uint32 div115k	Divisor for 115 kbauds generation
Uint32 divBR	Divisor for baud rate generation
Uint32 tcr	Transmission control register

Uint32 tlr	Trigger level register
Uint32 mdr1	Mode definition register 1
Uint32 mdr2	Mode definition register 2
Uint32 txfll	Transmit frame length register LSB
Uint32 txflh	Transmit frame length register MSB
Uint32 rxfll	Receive frame length register LSB
Uint32 rxflh	Receive frame length register MSB
Uint32 sflsr	Status FIFO line status register
Uint32 sfregl	Status FIFO register –LSB
Uint32 sfregh	Status FIFO register –MSB
Uint32 blr	Beginning of file length register
Uint32 pulse_width	Pulse width
Uint32 acreg	Auxiliary control register
Uint32 start_point	Start of IR
Uint32 rdrx	RX FIFO read pointer register
Uint32 wrrx	RX FIFO write pointer register
Uint32 rdtx	TX FIFO read pointer register
Uint32 wrtx	TX FIFO write pointer register
Uint32 rdst	Status FIFO read pointer register
Uint32 wrst	Status FIFO write pointer register

IRUART_DEVICE_CNT *Number of IRUART devices*

Constant IRUART_DEVICE_CNT

IRUART_DISABLE *Disable flag*

Constant IRUART_DISABLE

Description Used to disable a flag.

Example

```
IRUART_loopback(hIruArt,IRUART_DISABLE);
```

IRUART_ENABLE *Enable flag*

Constant IRUART_ENABLE

Description Used to enable a flag.

Example

```
IRUART_loopback(hIruArt,IRUART_ENABLE);
```

IRUART_OPEN_RESET *Flag used to reset an IRUART device while getting a handle*

Constant IRUART_OPEN_RESET

Description Used to reset an IRUART device after open.

Example

```
Handle hIruart;
...
hIruart = IRUART_open(IRUART_DEV0,IRUART_OPEN_RESET);
```

IRUART_Setup *IRUART set-up structure*

Structure IRUART_Setup

Members

Uint32 eFifoEnabled	Enable or disable FIFO. The constants for the flags are: ❑ IRUART_ENABLE ❑ IRUART_DISABLE

Uint32 eRxfifoTrigLevelStart	Trigger level to start transmission: ❏ IRUART_TRIG00 ❏ IRUART_TRIG04 ❏ IRUART_TRIG08 ❏ IRUART_TRIG12 ❏ IRUART_TRIG16 ❏ IRUART_TRIG20 ❏ IRUART_TRIG24 ❏ IRUART_TRIG28 ❏ IRUART_TRIG32 ❏ IRUART_TRIG36 ❏ IRUART_TRIG40 ❏ IRUART_TRIG44 ❏ IRUART_TRIG48 ❏ IRUART_TRIG52 ❏ IRUART_TRIG56 ❏ IRUART_TRIG60
Uint32 eRxfifoTrigLevelStop	Trigger level to stop transmission. Takes the same TRIG constants as trigger level to start transmission.
Uint32 eRxfifoTrigLevelInt	Trigger level to generate RHR1 interrupt. Takes the same TRIG constants as trigger level to start transmission.
Uint32 eTxfifoTrigLevelInt	Trigger level to generate THR interrupt. Takes the same TRIG constants as trigger level to start transmission.
Uint32 eStfifoTrigLevelInt	Trigger level to generate THR interrupt: ❏ IRUART_STS_TRIG0 ❏ IRUART_STS_TRIG4 ❏ IRUART_STS_TRIG7 ❏ IRUART_STS_TRIG8
Uint32 eWordLength	Word length 5, 6, 7, or 8. The constants are: ❏ IRUART_WORD8 ❏ IRUART_WORD7 ❏ IRUART_WORD6 ❏ IRUART_WORD5

Uint32 eParityEnable	Parity enable or disable. The constants for the flags are: ❑ IRUART_ENABLE ❑ IRUART_DISABLE
Uint32 eParity	Parity even or odd: ❑ IRUART_PARITY_ODD ❑ IRUART_PARITY_EVEN
Uint32 eStopBits	Number of stop bits. The constants are: ❑ IRUART_STOP1 ❑ IRUART_STOP1_PLUS_HALF ❑ IRUART_STOP2
Uint32 eBaudRate	Baud rate for receiving or transmission. In UART Mode Baud rate can take any value: ❑ IRUART_BAUD_115200 ❑ IRUART_BAUD_57600 ❑ IRUART_BAUD_38400 ❑ IRUART_BAUD_28800 ❑ IRUART_BAUD_19200 ❑ IRUART_BAUD_14400 ❑ IRUART_BAUD_9600 ❑ IRUART_BAUD_7200 ❑ IRUART_BAUD_4800 ❑ IRUART_BAUD_3600 ❑ IRUART_BAUD_2400 ❑ IRUART_BAUD_2000 ❑ IRUART_BAUD_1800 ❑ IRUART_BAUD_1200 ❑ IRUART_BAUD_600 ❑ IRUART_BAUD_300 In SIR Mode it is currently implemented to take: ❑ IRUART_BAUD_115200 ❑ IRUART_BAUD_57600 ❑ IRUART_BAUD_38400 ❑ IRUART_BAUD_19200 ❑ IRUART_BAUD_2400 ❑ IRUART_BAUD_9600 being default To support other bauds, either this function

	needs to be modified or use the IRUART_Config function where you can set your own bauds by setting registers div_115k and divBR
Uint32 eLoopBackEnable	Enable loopback or not. The constants for the flags are: ❑ IRUART_ENABLE ❑ IRUART_DISABLE
Uint32 swflowtype	Type of sw flow control
Uint32 eIntMask	Interrupt vector. The following flags can be ORed to produce the mask. In UART mode: ❑ IRUART_UART_INT_RHR – RHR interrupt ❑ IRUART_UART_INT_THR – THR interrupt ❑ IRUART_UART_INT_LINE_STS – line status interrupt ❑ IARUART_UART_INT_XOFF – XOFF interrupt In SIR mode: ❑ IRUART_SIR_INT_RHR – RHR interrupt ❑ IRUART_SIR_INT_THR – THR interrupt ❑ IRUART_SIR_INT_LAST_RXB – last byte in RX FIFO interrupt ❑ IRUART_SIR_INT_RX_OVER – RX overrun interrupt ❑ IRUART_SIR_INT_STS_FIFO – status FIFO interrupt ❑ IRUART_SIR_INT_TX_UNDER – TX underrun interrupt ❑ IRUART_SIR_INT_LINE_STS – line status interrupt ❑ IRUART_SIR_INT_EOF – EOF interrupt

Uint32 pulseWidth	Pulse width
Uint32 startIr	Start IR

IRUART_changeMode *Changes mode of IRUART device*

Function

```
void IRUART_changeMode(
    IRUART_Handle     hIruart,
    Uint32            flag
)
```

Arguments

hIruart — Device handle; see IRUART_open

flag — Mode to select. The flag takes values:
- IRUART_MODE_SIR
- IRUART_MODE_UART

Return Value none

Description Used to change the mode of the IRUART device.

Example

```
IRUART_changeMode(hIruart,IRUART_MODE_SIR);
```

IRUART_close *Closes IRUART device*

Function

```
void IRUART_close(
    IRUART_Handle     hIruart
)
```

Arguments

hIruart — Device handle; see IRUART_open

Return Value none

Description Closes a previously opened IRUART device (see IRUART_open). The following tasks are performed:

- The IRUART event is disabled and cleared.
- The IRUART registers are set to their default values.

Example

```
IRUART_close(hIruart);
```

IRUART_config *Sets IRUART configuration parameters*

Function

```
void IRUART_config(
    IRUART_Handle    hIruart,
    IRUART_Config    *config
)
```

Arguments

hIruart	Device handle; see IRUART_open
config	Pointer to the configuration structure

Return Value none

Description This is the IRUART configuration structure used to set up a IRUART device. You create and initialize this structure and then pass its address to the IRUART_config function.

Example

```
IRUART_Config MyConfig = {
  0x00000051u,          // fcr
  0x00000041u,          // scr
  0x00000003u,          // lcr
  0x00000000u,          // lsr
  0x00000002u,          // ssr
  0x00000040u,          // mcr
  0x00000000u,          // msr
  0x00000000u,          // ier
  0x0000003Eu,          // isr
  0x00000050u,          // efr
  0x00000000u,          // xon1
  0x00000000u,          // xon2
  0x00000000u,          // xoff1
  0x00000000u,          // xoff2
  0x00000000u,          // spr
  0x000001B2u,          // div115k
  0x00000001u,          // divBR
  0x00000080u,          // tcr
  0x00000000u,          // tlr
  0x00000000u,          // mdr1
  0x00000000u,          // mdr2
  0x00000000u,          // txfll
  0x00000000u,          // txflh
  0x00000000u,          // rxfll
  0x00000000u,          // rxflh
```

```
  0x00000000u,          // sflsr
  0x00000000u,          // sfregl
  0x00000000u,          // sfregh
  0x00000000u,          // blr
  0x00000000u,          // pulse_width
  0x00000000u,          // acreg
  0x00000000u,          // start_point
  0x00000000u,          // rdrx
  0x00000000u,          // wrrx
  0x00000000u,          // rdtx
  0x00000000u,          // wrtx
  0x00000000u,          // rdst
  0x00000000u,          // wrst
};
...
IRUART_config(hIruart, &MyConfig);
```

IRUART_eventDisable *Disables IRUART interrupts*

Function

```
void  IRUART_eventDisable(
    IRUART_Handle     hIruart,
    Uint32            eMask
)
```

Arguments

hIruart Device handle; see IRUART_open

eMask Mask specifies the events for which the interrupt is to be disabled. The following flags can be Ored to produce the mask.

In UART mode:

- IRUART_UART_INT_RHR – RHR interrupt
- IRUART_UART_INT_THR – THR interrupt
- IRUART_UART_INT_LINE_STS – line status interrupt
- IARUART_UART_INT_XOFF – XOFF interrupt

In SIR mode:

- IRUART_SIR_INT_RHR – RHR interrupt
- IRUART_SIR_INT_THR – THR interrupt
- IRUART_SIR_INT_LAST_RXB – last byte in RX FIFO interrupt
- IRUART_SIR_INT_RX_OVER – RX overrun interrupt
- IRUART_SIR_INT_STS_FIFO – status FIFO interrupt
- IRUART_SIR_INT_TX_UNDER – TX underrun interrupt

- IRUART_SIR_INT_LINE_STS – line status interrupt
- IRUART_SIR_INT_EOF – EOF interrupt

Return Value none

Description Used to disable the corresponding types of interrupts of IRUART device.

Example

```
IRUART_eventDisable(hIruart,IRUART_UART_INT_RHR |
IRUART_UART_INT_THR);
```

IRUART_eventEnable *Enables IRUART interrupts*

Function

```
void IRUART_eventEnable(
    IRUART_Handle    hIruart,
    Uint32           eMask
)
```

Arguments

hIruart Device handle; see IRUART_open

eMask Mask specifies the events for which the interrupt is to be enabled. The following flags can be Ored to produce the mask.

In UART mode:

- IRUART_UART_INT_RHR – RHR interrupt
- IRUART_UART_INT_THR – THR interrupt
- IRUART_UART_INT_LINE_STS – line status interrupt
- IARUART_UART_INT_XOFF – XOFF interrupt

In SIR mode:

- IRUART_SIR_INT_RHR – RHR interrupt
- IRUART_SIR_INT_THR – THR interrupt
- IRUART_SIR_INT_LAST_RXB – last byte in RX FIFO interrupt
- IRUART_SIR_INT_RX_OVER – RX overrun interrupt
- IRUART_SIR_INT_STS_FIFO – status fifo interrupt
- IRUART_SIR_INT_TX_UNDER – TX underrun interrupt
- IRUART_SIR_INT_LINE_STS – line status interrupt
- IRUART_SIR_INT_EOF – EOF interrupt

Return Value none

Description Used to enable the corresponding types of interrupts of IRUART device.

Example

```
IRUART_eventEnable(hIruart,IRUART_UART_INT_RHR |
IRUART_UART_INT_THR);
```

IRUART_fget — *Reads a character*

Function

```
Int32  IRUART_fget(
    IRUART_Handle    hIruart,
    char             *c
)
```

Arguments

hIruart	Device handle; see IRUART_open
c	Read character

Return Value Int32

Description Used to read one character from the IRUART device. Returns 1 if character is present in the incoming stream, –1 if there is no input available, or 2 when the character is the last character in a frame.

Example

```
char c;
flag = IRUART_fget(hIruart,&c);
if(flag) printf(”read char %d”,c);
```

IRUART_fgets — *Reads a string of characters*

Function

```
Uint32  IRUART_fgets(
    IRUART_Handle    hIruart,
    char             *buf,
    Uint32            nBytes
)
```

Arguments

hIruart	Device handle; see IRUART_open
buf	Character buffer
nBytes	Buffer length

Return Value Uint32

Description Used to read a string of characters from the IRUART device. Returns the character string appended with \0 or just null string (\0) if no data is present at the input. In SIR mode, it returns after an end of frame is detected or nBytes of characters are received, whichever comes first.

Example

```
char buf[30];
noofchar = IRUART_fgets(hIruart,buf,30);
while(buf[i]!=’\0’)
    printf(”read char %d”,buf[i++]);
```

IRUART_fput — *Writes a character*

Function

```
Int32  IRUART_fput(
    IRUART_Handle     hIruart,
    char              writechar
)
```

Arguments

hIruart — Device handle; see IRUART_open

writechar — Character to be written to output

Return Value

Int32

Description

Used to read one character from the IRUART device. Returns 1 if character can be written to the output stream or –1 if character cannot be written.

Example

```
char c=32;
flag = IRUART_fput(hIruart,c);
```

IRUART_fputs — *Writes a string of characters*

Function

```
Uint32  IRUART_fputs(
    IRUART_Handle     hIruart,
    char              *buf
)
```

Arguments

hIruart — Device handle; see IRUART_open

buf — Character buffer

Return Value

Uint32

Description

Used to write a string of characters to the IRUART device. Returns the number of characters written.

Example

```
char buf[30];
noofchar = IRUART_fputs(hIruart,buf);
```

IRUART_getBaudRate *Gets the baud rate*

Function

```
Uint32  IRUART_getBaudRate(
    IRUART_Handle     hIruart
)
```

Arguments hIruart Device handle; see IRUART_open

Return Value Uint32

Description Used to get the baud rate at which the IRUART device is operating.

Example

```
int baud;
baud = IRUART_getBaudRate(hIruart);
```

IRUART_getConfig *Gets the IRUART configuration parameters*

Function

```
void  IRUART_getConfig(
    IRUART_Handle     hIruart,
    IRUART_Config     *config
)
```

Arguments

hIruart Device handle; see IRUART_open

config Pointer to the destination configuration structure

Return Value none

Description Gets configuration structure for the given IRUART device, which is already opened. The return structure can be modified and passed to IRUART_config function, if changes are to be made.

Example

```
IRUART_Config MyConfig;
IRUART_getConfig(hIruart, &MyConfig);
```

IRUART_getEventId *Gets the IRQ event of IRUART device*

Function

```
Uint32  IRUART_getEventId(
    IRUART_Handle     hIruart
)
```

Arguments hIruart Device handle; see IRUART_open

Return Value Uint32

Description Use this function to obtain the event ID for the IRUART device.

Example

```
UartEventID = IRUART_getEventId(hIruart);
IRQ_enable(UartEventID);
```

IRUART_getFrameStatus *Gets frame status*

Function

```
Uint32  IRUART_getFrameStatus(
    IRUART_Handle    hIruart,
    Uint32           *length
)
```

Arguments

hIruart	Device handle; see IRUART_open
length	Length of the frame, a return parameter

Return Value Uint32

Description Used to get the frame status of the last frame received.

Example

```
Uint32 length,status;
status = IRUART_getFrameStatus(hIruart,&length);
```

IRUART_getIntType *Gets the type of interrupt occurred*

Function

```
Uint32  IRUART_getIntType(
    IRUART_Handle    hIruart
)
```

Arguments

hIruart	Device handle; see IRUART_open

Return Value Uint32

Description Used to get the type of interrupt pending with the IRUART device.

Example

```
intype = IRUART_getIntType(hIruart);
```

IRUART_getSetup *Gets the initial set-up values*

Function

```
void  IRUART_getSetup(
    IRUART_Handle    hIruart,
    IRUART_Setup     *setup
)
```

Arguments

hIruart	Device handle
setup	Initialization structure

Return Value none

Description Gets initialization structure for the given UART device, which is already opened. The return structure can be modified and passed to IRUART_setup function, if changes are to be made.

Example

```
IRUART_getSetup(hIruart,&Mysetup);
```

IRUART_loopBack *Enables or disables the IRUART device in loopback mode*

Function

```
void IRUART_loopBack(
    IRUART_Handle    hIruart,
    Uint32           flag
)
```

Arguments

hIruart Device handle; see IRUART_open

flag Flag. The constants that can be set are:
- IRUART_ENABLE
- IRUART_DISABLE

Return Value none

Description To enable or disable IRUART to loopback mode, used for testing the IRUART device.

Example

```
IRUART_loopBack(hIruart,IRUART_ENABLE);
```

IRUART_open *Opens IRUART device*

Function

```
IRUART_Handle IRUART_open(
    Uint16        devNum,
    Uint16        eFlags
)
```

Arguments

devNum Specifies the device to be opened

eFlags Open flags:
- OPEN_RESET – resets the IRUART, opens in SIR mode default
- OPEN_RESET | IRUART_MODE_UART – resets device and opens in UART mode
- OPEN_RESET | IRUART_MODE_SIR – reset device and opens in SIR mode

Return Value IRUART_Handle Device handle
INV – open failed

Description Before IRDA UART can be used, it must be 'opened' using this function. Once opened it cannot be opened again until it is 'closed' (see IRUART_close). The return value is a unique device handle that is used in subsequent IRUART API calls. If the open fails, 'INV' is returned.

If the OPEN_RESET flag is specified, the UART device registers are set to their power-on defaults and any associated interrupts are disabled and cleared. IRUART is opened in UART mode or in SIR mode depending on the flag specified as IRUART_MODE_UART or IRUART_MODE_SIR (default is the SIR mode).

Example

```
IRUART_Handle hIruart;
...
hIruart = IRUART_open(IRUART_DEV0, OPEN_RESET | UART_MODE_SIR );
```

IRUART_read *Reads a buffer of characters*

Function

```
Uint32  IRUART_read(
    IRUART_Handle     hIruart,
    char              *buf,
    Uint32            nBytes
)
```

Arguments

hIruart	Device handle; see IRUART_open
buf	Buffer
nBytes	Number of bytes

Return Value Uint32

Description Used to read a buffer of characters from the IRUART device. Returns number of characters read from the incoming stream. In SIR mode, it returns after an end of frame is detected or nBytes of characters are received, whichever comes first.

Example

```
char buf[30];
noofchar = IRUART_read(hIruart,buf,30);
for(i=0;i < noofchar;i++)
    printf("read char %d",buf[i]);
```

IRUART_reset *Resets the IRUART device*

Function

```
void  IRUART_reset(
    IRUART_Handle     hIruart
)
```

Arguments

hIruart Device handle; see IRUART_open

Return Value

none

Description

Resets the IRUART device. Disables and clears the interrupt event and sets the IRUART registers to their power-on default values.

Example

```
IRUART_reset(hIruart);
```

IRUART_setBaudRate *Sets the baud rate*

Function

```
void  IRUART_setBaudRate(
    IRUART_Handle     hIruart,
    Uint32            eBaudRate
)
```

Arguments

hIruart Device handle; see IRUART_open

eBaudRate Baud rate to be set

In UART mode, baud rate can take any value:

- IRUART_BAUD_115200
- IRUART_BAUD_57600
- IRUART_BAUD_38400
- IRUART_BAUD_28800
- IRUART_BAUD_19200
- IRUART_BAUD_14400
- IRUART_BAUD_9600
- IRUART_BAUD_7200
- IRUART_BAUD_4800
- IRUART_BAUD_3600
- IRUART_BAUD_2400
- IRUART_BAUD_2000
- IRUART_BAUD_1800
- IRUART_BAUD_1200
- IRUART_BAUD_600
- IRUART_BAUD_300

In SIR mode, it is currently implemented to take:

- IRUART_BAUD_115200
- IRUART_BAUD_57600
- IRUART_BAUD_38400
- IRUART_BAUD_19200
- IRUART_BAUD_2400
- IRUART_BAUD_9600 being default

To support other bauds, either this function needs to be modified or use the IRUART_Config function where you can set your own bauds by setting registers div_115k and divBR.

Return Value none

Description Used to set the baud rate at which the IRUART device should operate. In SIR mode, only baud rates 115200, 57600, 38400, 19200, 9600, and 2400 are supported (2400 is the default).

Example

```
IRUART_setBaudRate(hIruart,IRUART_BAUD_115200);
```

IRUART_setup — *Sets up initial parameters*

Function

```
void  IRUART_setup(
    IRUART_Handle      hIruart,
    IRUART_Setup       *setup
)
```

Arguments

hIruart	Device handle
setup	Initilaization structure

Return Value none

Description This is the IRUART initialization structure used to set up a IRUART device. You can create and initialize this structure and then pass its address to the IRUART_setup function.

Example

```
IRUART_setup mySetup{
IRUART_ENABLE,         // fifo enabled
IRUART_TRIG00,         // rx fifo trigger level, start transmissions
IRUART_TRIG08,         // rx fifo trigger level, stop transmissions
IRUART_TRIG08,         // rx fifo trigger levels generate interrupts
IRUART_TRIG08,         // tx fifo trigger levels generate interrupts
IRUART_STS_TRIG4       // Status fifo trig register
IRUART_WORD8,          // word length
IRUART_ENABLE,         // eParity enable
IRUART_PARITY_ODD,     // eParity type
IRUART_STOP1,          // number of stop bits
IRUART_BAUD_115200,    // baud rate
IRUART_DISABLE,        // loopback enable
0,                     // type of sw flow
IRUART_UART_INT_RHR|IRUART_UART_INT_THR,  // interrupt mask
6,                     // pulse width
2,                     //start IR
};
IRUART_setup(hIruart,&mySetup);
```

IRUART_write *Writes a buffer of characters*

Function

```
Uint32  IRUART_write(
    IRUART_Handle     hIruart,
    char              *buf,
    Uint32            nBytes
)
```

Arguments

hIruart	Device handle; see IRUART_open
buf	Buffer
nBytes	Number of bytes

Return Value Uint32

Description Used to write a buffer of characters to the IRUART device. Returns number of character written to the outgoing stream.

Example

```
char buf[30]={20,30,....};
noofchar = IRUART_write(hIruart,buf,30);
```

11.3 Register and Field Names

Table 11–2. IRUART Module Register and Field Names

Register Name	Field Name(s)
IRUART_RHR1	RX_BI (R), RX_FE (R), RX_PE (R), RHR1 (R)
IRUART_RHR2	EOF (R), RHR2 (R)
IRUART_THR	THR (W)
IRUART_FCR	RXF_TR, TXF_TR, RXF_CL, TXF_CL, FIFO_EN
IRUART_SCR	FINIT_ST, FINIT, RXCTUP_EN, TX_E_CTL_IT, FPTRACEN
IRUART_LCR	BREAK_EN, PAR_T2, PAR_T1, PAR_EN, NB_STOP, C_LN
IRUART_LSR1	RX_FIFO_STS, TX_SR_E, TX_FIFO_E, RX_OE, RX_FIFO_E
IRUART_LSR2	THR_EMPTY (R), STS_FIFO_FULL (R), RX_LAST_BYTE (R), FRAME2LONG (R), ABORT (R), CRC (R), STS_FIFO_E (R), RX_FIFO_E (R)
IRUART_SSR	RX_CT_WUP_STS, TX_FIFO_FULL
IRUART_MCR	CLKSEL, TCR_TLR, XON_EN, MODE
IRUART_MSR	
IRUART_IER1	XOFF_IT (R), L_IT (R), TH_IT (R), RH_IT (R)
IRUART_IER2	EOF_IT (R), L_IT (R), TX_UNRUN (R), STF_TRIG (R), RX_OVRUN (R), LASTRX_IT (R), TH_IT (R), RH_IT (R)
IRUART_ISR1	FCR_M1, FCR_M2, IT_TYPE, IT_PENDING
IRUART_ISR2	EOF_IT (R), L_IT (R), TX_UNRUN (R), STF_TRIG (R), RX_OVRUN (R), LASTRX_IT (R), TH_IT (R), RH_IT (R)
IRUART_EFR	SP_CHAR, ENHANCED_EN, SW_FLOW
IRUART_XON1	XON_WORD1
IRUART_XON2	XON_WORD2
IRUART_XOFF1	XOFF_WORD1
IRUART_XOFF2	XOFF_WORD2
IRUART_SPR	SPR_WORD

Note: R = Read only; W = Write only; fields not marked are R/W

Table 11–2. IRUART Module Register and Field Names (Continued)

Register Name	Field Name(s)
IRUART_DIV115K	DIV_115K
IRUART_DIVBR	DIV_BITRATE
IRUART_TCR	RXF_TR_START, RXF_TR_HALT
IRUART_TLR	RXF_TR_RHR1, TXF_TR_THR
IRUART_MDR1	MODE_SELECT
IRUART_MDR2	STS_FIFO_TRIG
IRUART_TXFLL	TXFLL (W)
IRUART_TXFLH	TXFLH (W)
IRUART_RXFLL	RXFLL (W)
IRUART_RXFLH	RXFLH (W)
IRUART_SFLSR	OE_ERR (R), FR_L_ERR (R), ABORT_DETECT (R), CRC (R)
IRUART_SFREGL	SFREGL (R)
IRUART_SFREGH	SFREGH (R)
IRUART_BLR	STS_FIFO_RESET, BOF_TYPE, NB_XBOF
IRUART_PW	PW
IRUART_ACREG	SD_MODE, SCTX_EN, ABORT_EN, EOT_EN
IRUART_STPT	PS
IRUART_WRRX	RX_WRITE_PTR
IRUART_RDRX	RX_READ_PTR
IRUART_WRTX	TX_WRITE_PTR
IRUART_RDTX	TX_READ_PTR
IRUART_WRST	ST_WRITE_PTR
IRUART_RDST	ST_READ_PTR
IRUART_RESUME	DI (R)

Note: R = Read only; W = Write only; fields not marked are R/W

Chapter 12

KBIO Module

KBIO is a special device for Keyboard I/O operations. There are 16 configurable KBIO pins in which 15–8 are treated as rows and 7–0 as column lines. KBIO lines can also be used as normal pins.

Topic **Page**

12.1 Overview

Table 12–1. KBIO Descriptions

Syntax	Type	Description	Page
KBIO_Config	S	KBIO configuration structure.	12-3
KBIO_DEVICE_CNT	C	KBIO device count.	12-3
KBIO_OPEN_RESET	C	KBIO OPEN_RESET flag.	12-3
KBIO_Setup	S	KBIO set-up structure.	12-3
KBIO_clearPinDelta	F	Clears the pin outputs.	12-4
KBIO_close	F	Closes a previously opened KBIO device.	12-5
KBIO_config	F	Configures the KBIO using configuration structure.	12-5
KBIO_getConfig	F	Reads the current KBIO configuration values.	12-6
KBIO_getDirection	F	Gets the input/output direction of the KBIO pins.	12-6
KBIO_getEventId	F	Gets the interrupt request event ID for the given KBIO pin.	12-7
KBIO_getIrqMode	F	Reads the current IRQ configuration.	12-7
KBIO_getPinDelta	F	Detects changed pins.	12-8
KBIO_getSetup	F	Returns the set-up parameters for a KBIO pin.	12-8
KBIO_getStatus	F	Gets the enable/disable status of the KBIO pins.	12-9
KBIO_open	F	Opens a KBIO device.	12-9
KBIO_readColumn	F	Reads the bit values from the COLUMN lines.	12-10
KBIO_readRow	F	Reads the bit values from the ROW lines.	12-10
KBIO_reset	F	Resets a KBIO device that is already opened.	12-10
KBIO_setDirection	F	Sets the input/output direction of the KBIO pins.	12-11
KBIO_setIrqMode	F	Sets the IRQ triggering mode.	12-11
KBIO_setStatus	F	Enables/disables the KBIO pins.	12-12
KBIO_setup	F	Sets the parameters for a KBIO pin using the KBIO_Setup structure.	12-12
KBIO_writeColumn	F	Writes the bit values to the COLUMN lines.	12-13
KBIO_writeRow	F	Writes the bit values to the ROW lines.	12-13

Note: C = Constant; F = Function; S = Structure

12.2 API Reference

KBIO_Config — *KBIO configuratrion structure*

Structure KBIO_Config

Members

Uint32 ior	KBIO input/output register
Uint32 cior	KBIO configuration register
Uint32 irqA	KBIO interrupt request register A
Uint32 irqB	KBIO interrupt request register B
Uint32 ddior	KBIO delta detect register
Uint32 enr	KBIO mux select register

Description This is the KBIO configuration structure used to set up a KBIO device. User can create and initialize this structure and then pass its address to the KBIO_config function.

KBIO_DEVICE_CNT — *KBIO device count*

Constant KBIO_DEVICE_CNT

KBIO_OPEN_RESET — *KBIO OPEN_RESET flag*

Constant KBIO_OPEN_RESET

Description This flag is used while opening KBIO device. To open with reset use KBIO_OPEN_RESET, otherwise use 0.

Example see KBIO_open

KBIO_Setup — *KBIO set-up structure*

Structure KBIO_Setup

Members

Uint16 enab	KBIO pin enable/disable (multiplexing); see KBIO_setStatus

Uint16 dir	KBIO input/output direction see L
Uint16 irqMode	KBIO interrupt trigger mode see L

Description

This is the KBIO set-up structure used to set up a KBIO pin. User can create and initialize this structure and then pass its address to the KBIO_setup function with pinID.

KBIO_clearPinDelta *Clears pin outputs*

Function

```
void  KBIO_clearPinDelta(
    KBIO_Handle       hKBIO,
    Uint32            pinMask
)
```

Arguments

hKBIO	Device handle
pinMask	KBIO pin mask

Return Value

none

Description

Used to clear bits of given input pins in delta detect register. Available pin IDs are as follows (to get pinMask, user can OR them for grouping pins):

- KBIO_PIN0
- KBIO_PIN1
- KBIO_PIN2
- KBIO_PIN3
- KBIO_PIN4
- KBIO_PIN5
- KBIO_PIN6
- KBIO_PIN7
- KBIO_PIN8
- KBIO_PIN9
- KBIO_PIN10
- KBIO_PIN11
- KBIO_PIN12
- KBIO_PIN13
- KBIO_PIN14
- KBIO_PIN15
- KBIO_PINROW
- KBIO_PINCOL

Example

```
KBIO_clearPinDelta(hKBIO, 0x005FF0);
```

KBIO_close — *Closes previously opened KBIO device*

Function

```
void  KBIO_close(
    KBIO_Handle       hKBIO
)
```

Arguments

hKBIO Device handle; see KBIO_open

Return Value

none

Description

Closes a previously opened KBIO device (see KBIO_open). The KBIO registers are set to their default values.

Example

```
KBIO_close(hKBIO);
```

KBIO_config — *Configures KBIO using configuration structure*

Function

```
void  KBIO_config(
    KBIO_Handle       hKBIO,
    KBIO_Config       *myConfig
)
```

Arguments

hKBIO Device handle; see KBIO_open

myConfig Pointer to the configuration structure

Return Value

none

Description

Sets up the KBIO device using the configuration structure. The values of the structure are written to the KBIO registers.

Example

```
KBIO_Config MyConfig = {
  0x0u,                    // ior
  0x000FFFFFu              // cior
  0x0u,      // irqA
  0x0u,      // irqB
  0x000FFFFFu,    // ddior
  0x000FFFFFu     // enr
};
...
KBIO_config(hKBIO, &MyConfig);
```

KBIO_getConfig — *Reads the current KBIO configuration values*

Function

```
void  KBIO_getConfig(
    KBIO_Handle       hKBIO,
    KBIO_Config       *config
)
```

Arguments

KBIO	Device handle; see KBIO_open
config	Pointer to the destination configuration structure

Return Value none

Description Gets the current KBIO configuration values.

Example

```
KBIO_Config kbioCfg;
KBIO_getConfig(hKBIO, &kbioCfg);
```

KBIO_getDirection — *Gets input/output direction of the KBIO pins*

Function

```
Uint32  KBIO_getDirection(
    KBIO_Handle       hKBIO,
    Uint32            pinMask
)
```

Arguments

hKBIO	Device handle; see KBIO_open
pinMask	I/O pin mask

Return Value Uint32

Description Use this function to get the input/output direction of the pins specified in pinMask. See KBIO_clearPinDelta for pinMask specification dir – KBIO_IN, KBIO_OUT.

Example

```
PinMaskDir = KBIO_getDirection(hKBIO, KBIO_PINROW);
```

KBIO_getEventId — *Gets interrupt request event ID for the given KBIO pin*

Function

```
Uint16  KBIO_getEventId(
    KBIO_Handle      hKBIO,
    Uint32           ePinId
)
```

Arguments

hKBIO	Device handle; see KBIO_open
ePinId	KBIO pin ID

Return Value

Uint16	IRQ event ID for the KBIO device

Description

Use this function to obtain the event ID for the KBIO device. See KBIO_clear-PinDelta for available pin IDs. Return values:

- IRQ_EVT_KBIO_COL – KBIO row pins (8–15)
- IRQ_EVT_KBIO_ROW – KBIO column pins (0–7)
- 0xFFFF – ERROR

Example

```
KBIOEventID = KBIO_getEventId(hKBIO, KBIO_PINROW);
IRQ_enable(KBIOEventID);
```

KBIO_getIrqMode — *Reads current IRQ configuration*

Function

```
Uint16  KBIO_getIrqMode(
    KBIO_Handle      hKBIO,
    Uint32           ePinId
)
```

Arguments

hKBIO	Device handle
ePinId	KBIO pin ID

Return Value

Uint16	Current IRQ configuration

Description

Use this function to get the IRQ configuration. Return values:

- 0 – KBIO_IRQ_DIS (disable IRQ)
- 1 – KBIO_IRQ_RISE (IRQ generated on rising edge)
- 2 – KBIO_IRQ_FALL (IRQ generated on falling edge)
- 3 – KBIO_IRQ_STCH (IRQ generated on state change)

Example

```
IrqMode = KBIO_getIrqMode (hKBIO, KBIO_PIN3);
```

KBIO_getPinDelta *Detects changed pins*

Function

```
Uint32  KBIO_getPinDelta(
    KBIO_Handle       hKBIO,
    Uint32            pinMask
)
```

Arguments

hKBIO	Device handle
pinMask	KBIO pin mask

Return Value Uint32

Description Use this function to read the change in the pins specified by pinMask. See KBIO_clearPinDelta for pinMask specification.

Example

```
deltaPattern = KBIO_getPinDelta(hKBIO, KBIO_PINROW);
```

KBIO_getSetup *Returns the set-up parameters for a KBIO pin*

Function

```
void  KBIO_getSetup(
    KBIO_Handle       hKBIO,
    Uint32            ePinId,
    KBIO_Setup        *setup
)
```

Arguments

hKBIO	Device handle; see KBIO_open
ePinId	KBIO pin ID
setup	Set-up structure

Return Value none

Description Return the setup values used for a specified pin. PinMask cannot be used for this API. See KBIO_clearPinDelta for available pin IDs. See KBIO_Setup for setup parameters.

Example

```
KBIO_Setup MySetup;
...
KBIO_getSetup(hKBIO, KBIO_PIN0, &MySetup);
```

KBIO_getStatus — *Gets enable/disable status of the KBIO pins*

Function

```
Uint32  KBIO_getStatus(
    KBIO_Handle     hKBIO,
    Uint32          pinMask
)
```

Arguments

hKBIO	Device handle; see option
pinMask	I/O pin mask

Return Value Uint32

Description Use this function to get the enable disable status of the KBIO pins specified by pinMask. See KBIO_clearPinDelta for pinMask specification modes.

- KBIO_ENABLE – KBIO enable
- KBIO_DISABLE – KBIO disable, configuration for other I/O signal

Example

```
PinStatus = KBIO_getStatus(hKBIO, KBIO_PINROW);
```

KBIO_open — *Opens KBIO Device*

Function

```
KBIO_Handle  KBIO_open(
    Uint16          devNum,
    Uint16          Flags
)
```

Arguments

devNum	specifies the device to be opened
Flags	Open flags // + KBIO_OPEN_RESET – resets the KBIO

Return Value KBIO_Handle Device handle

Description Before a KBIO can be used, it must be ‘opened’ using this function. Once opened it cannot be opened again until it is ‘closed’ The return value is a unique device handle that is used in subsequent KBIO API calls. If the open fails, ‘INV’ is returned. If the KBIO_OPEN_RESET flag is specified, the KBIO device registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example

```
KBIO_Handle hKBIO;
hKBIO = KBIO_open(KBIO_DEV0, KBIO_OPEN_RESET);
```

KBIO_readColumn *Reads the bit values from the COLUMN lines*

Function

```
Uint32  KBIO_readColumn(
    KBIO_Handle      hKBIO
)
```

Arguments hKBIO KBIO device handle

Return Value Uint32

Description Used to read the bit values from the COLUMN lines (ior, 7–0). Used for KBIO only.

Example

```
Uint32 colVal = KBIO_readColumn(hKBIO);
```

KBIO_readRow *Reads the bit values from the ROW lines*

Function

```
Uint32  KBIO_readRow(
    KBIO_Handle      hKBIO
)
```

Arguments hKBIO KBIO device handle

Return Value Uint32

Description Used to read the bit values from the ROW lines (ior, 15–8). Used for KBIO only.

Example

```
Uint32 rowVal = KBIO_readRow(hKBIO);
```

KBIO_reset *Resets KBIO Device that is already opened*

Function

```
void  KBIO_reset(
    KBIO_Handle      hKBIO
)
```

Arguments hKBIO Device handle; see KBIO_open

Return Value none

Description Disables and clears the interrupt event and sets the KBIO registers to their default values.

Example

```
KBIO_reset(hKBIO);
```

KBIO_setDirection *Sets input/output direction of the KBIO pins*

Function

```
void  KBIO_setDirection(
    KBIO_Handle      hKBIO,
    Uint32           pinMask,
    Uint16           dir
)
```

Arguments

hKBIO	Device handle; see KBIO_open
pinMask	I/O pin mask
dir	Input/output direction

Return Value

none

Description

Use this function to set the input/output direction of the pins specified in pinMask. See KBIO_clearPinDelta for pinMask specification dir – KBIO_IN, KBIO_OUT.

Example

```
KBIO_setDirection(hKBIO, KBIO_PINROW, KBIO_IN);
```

KBIO_setIrqMode *Sets IRQ triggering mode*

Function

```
void  KBIO_setIrqMode(
    KBIO_Handle      hKBIO,
    Uint32           pinMask,
    Uint16           eMode
)
```

Arguments

hKBIO	Device handle
pinMask	KBIO pin mask
eMode	IRQ modes (enumerated)

Return Value

none

Description

Enables/disables the interrupts in the pins specified by the pinMask to rising edge, falling edge, or on state change according to the eMode.

Example

```
KBIO_setIrqMode (hKBIO, KBIO_PINROW, KBIO_IRQ_FALL);
```

KBIO_setStatus — *Enable/disable the KBIO pins*

Function

```
void  KBIO_setStatus(
    KBIO_Handle      hKbio,
    Uint32           pinMask,
    Uint16           mode
)
```

Arguments

hKBIO	Device handle; see option
pinMask	I/O pin mask
mode	Enable/disable mode

Return Value none

Description Use this function to set the enable/disable status of the KBIO pins specified by pinMask. See KBIO_clearPinDelta for pinMask specification modes.

- KBIO_ENABLE – KBIO enable
- KBIO_DISABLE – KBIO disable, configuration for other I/O signal

Example

```
PinStatus = KBIO_setStatus(hKBIO, KBIO_PINROW, KBIO_ENABLE);
```

KBIO_setup — *Sets the parameters for a KBIO pin using the KBIO_Setup structure*

Function

```
void  KBIO_setup(
    KBIO_Handle      hKBIO,
    Uint32           ePinId,
    KBIO_Setup       *MySetup
)
```

Arguments

hKBIO	Device handle
ePinId	KBIO pins
MySetup	Set-up structure

Return Value none

Description This API is used to set up KBIO pins. User can create and initialize set-up structure and then pass its address to the setup function with pin ID. See KBIO_clearPinDelta for available pin IDs.

Example

```
Setup MySetup = {
     1,
     1,
     KBIO_IRQ_RISE
};
...
KBIO_setup(hKBIO, KBIO_PINROW, &MySetup);
```

KBIO_writeColumn *Writes the bit values to the COLUMN lines*

Function

```
void KBIO_writeColumn(
   KBIO_Handle    hKBIO,
   Uint16         bitPattern
)
```

Arguments

hKBIO	KBIO device handle
bitPattern	Column bit pattern (8 bits only)

Return Value none

Description Used to write the bit values to the COLUMN lines (ior, 7–0). Used for KBIO only.

Example

```
KBIO_writeColumn(hKBIO, 0xFF);
```

KBIO_writeRow *Writes the bit values to the ROW lines*

Function

```
void KBIO_writeRow(
   KBIO_Handle    hKBIO,
   Uint16         bitPattern
)
```

Arguments

hKBIO	KBIO device handle
bitPattern	ROW bit pattern (8 bits only)

Return Value none

Description Used to write the bit values to the ROW lines (ior, 15–8). Used for KBIO only.

Example

```
KBIO_writeRow(hKBIO, 0xFFu);
```

12.3 Register and Field Names

Table 12–2. KBIO Module Register and Field Names

Register Name	Field Name(s)
KBIO_IOR	ROW, COL
KBIO_CIOR	ROW, COL
KBIO_IRQA	ROW, COL
KBIO_IRQB	ROW, COL
KBIO_DDIOR	ROW, COL
KBIO_ENR	ROW, COL

Note: All fields are Read/Write

Chapter 13

SDRAM Module

The SDRAM module provides functions and macros for configuring and initializing the on-chip SDRAM memory interface module. The SDRAM interface module effectively sits between the ARM processor and the SDRAM controller and functions as an isolation module between the two. It operates with the MCU memory interface so that SDRAM memories can be used on the same board with Flash and/or SRAM.

Topic **Page**

13.1 Overview

Table 13–1. SDRAM Descriptions

Syntax	Type	Description	Page
SDRAM_Config	S	SDRAM configuration structure.	13-3
SDRAM_Setup	S	SDRAM set-up structure.	13-3
SDRAM_config	F	Configures the SDRAM using the configuration structure.	13-4
SDRAM_getConfig	F	Retrieves the SDRAM register values.	13-5
SDRAM_getSetup	F	Gets the SDRAM configuration.	13-5
SDRAM_setup	F	Sets up the SDRAM using the set-up structure.	13-6

Note: F = Function; S = Structure

13.2 API Reference

SDRAM_Config — *SDRAM configuration structure*

Structure SDRAM_Config

Members

Uint32 dbcr	SDRAM data bus size control register
Uint32 cfgr	SDRAM configuration register
Uint32 rfcr	SDRAM refresh counter register
Uint32 ctlr	SDRAM control register
Uint32 ircr	SDRAM initialization refresh counter register

Description This is the SDRAM configuration structure used to set up the SDRAM memory interface. In order to configure the SDRAM, the SDRAM_Config structure is initialized with the SDRAM register values and its address is passed to the SDRAM_config function.

SDRAM_Setup — *SDRAM set-up structure*

Structure SDRAM_Setup

Members

Uint16 dataWidth	Device bus size
Bool endianness	Big or little endian
Uint16 dummyCycles	Dummy cycles to be inserted during bank switching
Uint16 dummyCyclesMatrix	Bank-switching dummy-cycle matrix
Uint16 casLatency	CAS latency in SDRAM clock cycles
Uint16 trcLatency	TRC latency in SDRAM clock cycles
Uint16 trpLatency	TRP latency in SDRAM clock cycles
Uint16 rasLatency	RAS latency in SDRAM clock cycles

Bool selfRefresh	Self-refresh request
Uint16 memoryWidth	Memory width – 16-/32-bit wide
Bool clockEnable	Clock disabled/enabled
Bool optimization	SDRAM controller optimization – ON/OFF
Bool clockDivider	Clock division
Uint16 numBanks	Number of banks – 2 or 4
Uint16 memorySize	Size of the SDRAM memory – 16/64/128/256M bits
Uint16 refreshCounter	Refresh counter value in SDRAM clock cycles
Uint16 predivider	SDRAM refresh counter predivider
Uint16 nopCycleCount	NOP command cycle count
Uint16 refreshCycleCount	No. of autorefresh cycles in IDLE state

Description

This is the SDRAM set-up structure which is used to configure the SDRAM memory interface using the SDRAM_setup function. The SDRAM_Setup structure is to be initialized with the required parameters before calling the SDRAM_setup function.

SDRAM_config *ConfigureS the SDRAM using the configuration structure*

Function

```
void  SDRAM_config(
    SDRAM_Config      *config
)
```

Arguments

config — The initialized SDRAM configuration structure

Return Value

none

Description

Configures the SDRAM memory interface using SDRAM device register values passed in through the SDRAM_Config structure. The register values could be retrieved by calling the SDRAM_getConfig function.

Example

```
SDRAM_Config myConfig = {
  0x00000002, // SDRAM_REG
  0x000006a7, // SDRAM_CONFIG
  0x00000320, // SDRAM_REF_COUNT
  0x00000003, // SDRAM_CNTL
  0x00021388  // SDRAM_INIT_CONF
};
SDRAM_config(&myConfig);
```

SDRAM_getConfig *Retrieves the SDRAM register values*

Function

```
void  SDRAM_getConfig(
    SDRAM_Config      *config
)
```

Arguments

config   The SDRAM config structure

Return Value

none

Description

This function returns the SDRAM device register values in a SDRAM_Config structure that is passed to it as an argument.

Example

```
SDRAM_Config myConfig;
SDRAM_getConfig(&myConfig);
```

SDRAM_getSetup *Gets the SDRAM configuration*

Function

```
void  SDRAM_getSetup(
    SDRAM_Setup       *setup
)
```

Arguments

setup   The SDRAM set-up structure

Return Value

none

Description

This function returns the SDRAM memory interface's current configuration in the SDRAM_Setup structure that is passed to it as an argument.

Example

```
SDRAM_Setup mySetup;
SDRAM_getSetup(&mySetup);
```

SDRAM_setup *Sets up the SDRAM using the set-up structure*

Function

```
void  SDRAM_setup(
    SDRAM_Setup       *setup
)
```

Arguments

setup — The SDRAM setup structure; see SDRAM_setupArgs for the values that the members of the setup structure can take

Return Value

none

Description

Configures the SDRAM memory interface using set-up parameters passed in through the SDRAM_Setup structure. The function kick starts the SDRAM initialization procedure after the registers have been appropriately programmed.

Example

```
SDRAM_Setup mySetup = {
  SDRAM_DATAWIDTH_32,   // Device bus size
  SDRAM_ENDIAN_LITTLE,  // Big or little endian
  0,                    // Dummy cycles to be inserted during bank switching
  0,                    // Bank-switching dummy-cycle matrix
  SDRAM_LATENCY_17,     // TCAS latency in SDRAM clock cycles
  SDRAM_LATENCY_6,      // TRC latency in SDRAM clock cycles
  SDRAM_LATENCY_3,      // TRP latency in SDRAM clock cycles
  SDRAM_LATENCY_5,      // TRAS latency in SDRAM clock cycles
  SDRAM_SELFREFRESH_OFF,// Self-refresh request
  SDRAM_MEMWIDTH_32,    // Memory width : 16/32 bit wide
  SDRAM_CLOCK_ENABLED,  // Clock enable: read-only bit; has no effect
  SDRAM_OPT_ON,         // SDRAM Controller optimization : ON/OFF
  SDRAM_CLKDIV_OFF,     // Clock Division
  SDRAM_NUMBANKS_4,     // Number of banks : 2 or 4
  SDRAM_MEMSIZE_64M,    // Size of the SDRAM memory - 16/64/128/256M bits
  800,                  // Refresh counter value in SDRAM clock cycles
  SDRAM_PREDIV_BY1,     // SDRAM refresh counter predivider
  5000,                 // NOP command cycle-count
  8                     // No. of autorefresh cycles in IDLE state
};
SDRAM_setup(&mySetup);
```

13.3 Register and Field Names

Table 13–2. SDRAM Module Register and Field Names

Register Name	Field Name(s)
SDRAM_DBCR	BIGEND, DC, DVS
SDRAM_CFGR	SD_SIZE, SD_BANK, SD_CDIV. OPT_ON_OFF, SD_CKE, SDRAM_32_BIT, SD_SLFR, SD_TRAS, SD_TRP, SD_TRC, SD_TCAS
SDRAM_RFCR	DIVIDER, REFRESH_COUNTER
SDRAM_CTLR	READY (R), SDRAM_INIT
SDRAM_IRCR	INIT_REF_MAX_CNT, INIT_NOP_MAX_CNT

Note: R = Read only; fields not marked are Read/Write

Chapter 14

SPI Module

The SPI module provides functions and macros for configuring and controlling the on-chip Serial Port Interface (SPI) device. The serial interface is a bi-directional three-line interface dedicated to the transfer of data to and from external devices using a three-line serial interface. The SPI interface is fully duplexed and is configurable from 1 to 32 bits, providing three enable signals programmable either as positive or negative edge- or level-sensitive.

Topic **Page**

14.1 Overview

Table 14–1. SPI Descriptions

Syntax	Type	Description	Page
SPI_Config	S	SPI configuration structure.	14-3
SPI_DEVICE_CNT	C	The number of SPI devices.	14-3
SPI_OPEN_RESET	C	Optional flag for SPI_open.	14-3
SPI_Setup	S	SPI set-up structure.	14-4
SPI_chkReadEnd	F	Checks for completion of the previous read operation.	14-5
SPI_chkWriteEnd	F	Checks for completion of the previous write operation.	14-5
SPI_close	F	Closes a previously opened SPI device.	14-6
SPI_config	F	Configures the SPI port using configuration structure.	14-6
SPI_eventDisable	F	Disables interrupt generation for the specified events.	14-7
SPI_eventEnable	F	Enables interrupt generation for the specified events.	14-7
SPI_getConfig	F	Returns the SPI register values.	14-8
SPI_getEventId	F	Gets the SPI interrupt event ID.	14-8
SPI_getSetup	F	Returns the SPI set-up parameters.	14-9
SPI_open	F	Opens the SPI port for use.	14-9
SPI_readWord	F	Reads in a data word from an SPI device.	14-10
SPI_reset	F	Resets the SPI device.	14-11
SPI_setup	F	Configures the SPI device through a set-up structure.	14-11
SPI_writeWord	F	Sends a data word through an SPI device.	14-12

Note: C = Constant; F = Function; S = Structure

14.2 API Reference

SPI_Config — *SPI configuration structure*

Structure SPI_Config

Members

Uint32 ssr	SPI setup register
Uint32 scr	SPI control register
Uint32 str	SPI status register; read only, value ignored by SPI_config
Uint32 txr	SPI transmit register; value ignored by SPI_config
Uint32 rxr	SPI receive register; read only, value ignored by SPI_config

Description This is the SPI configuration structure used to set up an SPI device. In order to configure SPI, the user is to create and initialize the structure with appropriate register values and pass its address to the SPI_config function.

SPI_DEVICE_CNT — *The number of SPI devices*

Constant SPI_DEVICE_CNT

SPI_OPEN_RESET — *Optional flag for SPI_open*

Constant SPI_OPEN_RESET

SPI_Setup *SPI set-up structure*

Structure SPI_Setup

Members

Uint16 dev0params — SPI device 0 parameters. Could be an ORed combination of a flag specifiying the format of the enable signal and a flag specifying the format of th clock signal.The flags that could be used to specify the format of the enable signal are:

- SPI_EN_POSITIVE_LEVEL
- SPI_EN_NEGATIVE_LEVEL
- SPI_EN_POSITIVE_EDGE
- SPI_EN_NEGATIVE_EDGE

The flags used to specify the format of the clock signal are:

- SPI_CLOCK_FALLING_EDGE
- SPI_CLOCK_RISING_EDGE

Uint16 dev1params — SPI device 1 parameters

Uint16 dev2params — SPI device 2 parameters

Uint16 eventMask — Specifies the events for which an SPI interrupt must be generated. Could be an ORed combination of one or more of the following flags:

- SPI_INTERRUPT_READ – read/write cycle
- SPI_INTERRUPT_WRITE – write cycle

Uint16 prescaler — The prescale clock divisor

Description This is the SPI set-up structure used to configure an SPI device. In order to configure the SPI, the user is to create and initialize the structure with appropriate parameter values and pass its address to the SPI_setup function.

SPI_chkReadEnd — *Checks for completion of the previous read operation*

Function

```
Bool SPI_chkReadEnd(
    SPI_Handle        hSPI
)
```

Arguments

hSPI	Device handle; see SPI_open

Return Value

Bool — Status of read process:

- TRUE: loading of SPI receive register is complete
- FALSE: loading of SPI receive register is not complete

Description

This function reads the RE (read-end) bit in the SPI status register to check for the completion of a read process. The function returns TRUE if the loading of the SPI receive register with the specified number of bits has been completed.

Example

```
while(!SPI_chkReadEnd(hSPI)); // wait for completion of read
```

SPI_chkWriteEnd — *Checks for completion of the previous write operation*

Function

```
Bool SPI_chkWriteEnd(
    SPI_Handle        hSPI
)
```

Arguments

hSPI	Device handle; see SPI_open

Return Value

Bool — Status of write process:

- TRUE: serialization is complete
- FALSE: serializatoin is not complete

Description

This function reads the WE (write-end) bit in the SPI status register to check for the completion of a write process. The function returns TRUE if the serialization of the last word sent has been completed.

Example

```
SPI_write(
  hSPI,      // handle to SPI
  SPI_DEV2,  // send to device 2
  16,        // data-size: 16bits
  0xC0DE     // data-word to send
);
while(!SPI_chkWriteEnd(hSPI)); // wait for completion
```

SPI_close — *Closes previously opened SPI device*

Function

```
void  SPI_close(
    SPI_Handle        hSPI
)
```

Arguments

hSPI — Device handle; see SPI_open

Return Value

none

Description

Closes a previously opened SPI device (see SPI_open).

Example

```
SPI_close(hSPI);
```

SPI_config — *Configures SPI port using configuration structure*

Function

```
void  SPI_config(
    SPI_Handle        hSPI,
    SPI_Config        *config
)
```

Arguments

hSPI — Device handle; see SPI_open

config — Pointer to the configuration structure

Return Value

none

Description

SPI_config sets up the SPI device using the configuration structure passed to it as parameter. The funciton writes the values of the structure's members to the corresponding SPI registers.

Example

```
SPI_Config myConfig = {
  0x00002B46,
  0x000000A8,
  0,
  0,
  0
};
SPI_config(hSPI, &myConfig);
```

SPI_eventDisable *Disables interrupt generation for the specified events*

Function

```
void  SPI_eventDisable(
    SPI_Handle      hSPI,
    Uint16          eventMask
)
```

Arguments

hSPI — Device handle; see SPI_open

eventMask — Mask specifying the events for which interrupt generation is to be disabled. Could be an ORed combination of the following flags:
- SPI_INTERRUPT_READ
- SPI_INTERRUPT_WRITE

Return Value none

Description This function disables the generation of an SPI interrupt for the specified events. The SPI supports interrupt generation for two events: the completion of a read cycle and the completion of a write cycle.

Example

```
SPI_eventDisable(hSPI, SPI_INTERRUPT_WRITE);// disable interrupt for write cycle
```

SPI_eventEnable *Enables interrupt generation for the specified events*

Function

```
void  SPI_eventEnable(
    SPI_Handle      hSPI,
    Uint16          eventMask
)
```

Arguments

hSPI — Device handle; see SPI_open

eventMask — Mask specifying the events for which interrupt generation is to be enabled. Could be an ORed combination of the following flags:
- SPI_INTERRUPT_READ
- SPI_INTERRUPT_WRITE

Return Value none

Description This function enables the generation of an SPI interrupt for the specified events. The SPI supports interrupt generation for two events: the completion of a read cycle and the completion of a write cycle.

Example

```
SPI_eventEnable(hSPI, SPI_INTERRUPT_READ);// enable interrupt for read/write cycle
```

SPI_getConfig — *Returns the SPI register values*

Function

```
void  SPI_getConfig(
    SPI_Handle        hSPI,
    SPI_Config        *config
)
```

Arguments

hSPI	Device handle; see SPI_open
config	Pointer to the destination configuration structure

Return Value none

Description This function retrieves the SPI register values in the SPI_Config structure passed to it as parameter.

Example

```
SPI_Config myConfig;
SPI_getConfig(hSPI, &myConfig);
```

SPI_getEventId — *Gets the SPI interrupt event ID*

Function

```
Uint16  SPI_getEventId(
    SPI_Handle        hSPI
)
```

Arguments

hSPI	Device handle; see SPI_open

Return Value

Uint16	The associated event ID

Description SPI_getEventId returns the event ID of the interrupt associated with the SPI device.

Example

```
Uint16 evt;
evt = SPI_getEventId(hSPI);
IRQ_enable(evt);
```

SPI_getSetup

Returns the SPI set-up parameters

Function

```
void  SPI_getSetup(
    SPI_Handle       hSPI,
    SPI_Setup        *setup
)
```

Arguments

hSPI	Device handle; see SPI_open
setup	SPI_Setup structure

Return Value

none

Description

This function returns the SPI's current configuration in the SPI_Setup structure that is passed to it as an argument.

Example

```
SPI_Setup mySetup;
SPI_getSetup(&mySetup);
```

SPI_open

Opens SPI port for use

Function

```
SPI_Handle  SPI_open(
    Uint16          devNum,
    Uint16          flags
)
```

Arguments

devNum — Device Number:
- DEVANY – open any available device
- DEV0 – open device 0

flags — Open flags
- OPEN_RESET – resets the SPI

Return Value

SPI_Handle — Device handle
- INV – open failed

Description

Before a SPI port can be used, it must be 'opened' using this function. Once opened it cannot be opened again until it is 'closed' (see SPI_close). The SPI_open function returns a unique device handle that is to be used in subsequent SPI calls. If SPI_open fails, it returns 'INV'.

If the SPI_OPEN_RESET flag is specified, the timer device registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example

```
Handle hSPI;
...
hSPI = SPI_open(SPI_DEVANY, SPI_OPEN_RESET);
```

SPI_readWord *Reads in a data word from an SPI device*

Function

```
Uint32  SPI_readWord(
    SPI_Handle     hSPI,
    Uint16         devNum,
    Uint16         nBits,
    Uint32         dataWrite
)
```

Arguments

hSPI	Device handle; see SPI_open
devNum	The device number: ❑ SPI_DEV0 ❑ SPI_DEV1 ❑ SPI_DEV2
nBits	Word size: between 1 and 32
dataWrite	The data to concurrently written

Return Value

Uint32	The read-in data word

Description

The SPI_read function reads in a 'word' of data from the specified SPI device. The function takes in, as parameters, the word size in bits (1 to 32) and the device number from which to read. An SPI read process is always simultaneous with a write process, hence, the function also takes in a (least-significant bit-aligned) data for the concurrent data write. The user may pass a dummy data value, if no data is to be transmitted. The data word read in is aligned to fit into the least-significant bits, although the SPI device receives the data aligned to the most-significant bits. The function returns only when the specified number of bits have been read.

Example

```
data_in = SPI_readWord(
  hSPI,
  SPI_DEV2, // use device 2
  16,       // no. of bits to send
  0         // dummy data for the concurrent write
);
```

SPI_reset — *Resets the SPI device*

Function

```
void SPI_reset(
    SPI_Handle      hSPI
)
```

Arguments

hSPI Device handle; see SPI_open

Return Value

none

Description

This function resets the SPI device setting all device registers to their power-on defaults. It also disables and clears the interrupt associated with the device.

Example

```
SPI_reset(hSpi);
```

SPI_setup — *Configures SPI device through a set-up structure*

Function

```
void SPI_setup(
    SPI_Handle      hSPI,
    SPI_Setup       *setup
)
```

Arguments

hSPI Device handle; see SPI_open

setup The initialized SPI_Setup structure

Return Value

none

Description

This function sets up the SPI device using the parameters passed in through the SPI_Setup structure.

Example

```
SPI_Setup mySetup = {
  SPI_EN_POSITIVE_LEVEL | SPI_CLOCK_RISING_EDGE,
  SPI_EN_NEGATIVE_EDGE | SPI_CLOCK_FALLING_EDGE,
  SPI_EN_POSITIVE_EDGE | SPI_CLOCK_RISING_EDGE,
  SPI_INTERRUPT_READ | SPI_INTERRUPT_WRITE,
  SPI_PRESCALE_BY1
};
SPI_setup(hSPI, &mySetup);
```

SPI_writeWord — *Sends a data word through an SPI device*

Function

```
void  SPI_writeWord(
    SPI_Handle      hSPI,
    Uint16          devNum,
    Uint16          nBits,
    Uint32          dataWrite
)
```

Arguments

hSPI	Device handle; see SPI_open
devNum	The device number: ❑ SPI_DEV0 ❑ SPI_DEV1 ❑ SPI_DEV2
nBits	Word size: between 1 and 32
dataWrite	The data to be written

Return Value

none

Description

The SPI_write writes a 'word' of data to the specified SPI device. The function takes in, as parameters, the word size in bits (1 to 32) and the device number to write the data to. Note that the data to be sent must be aligned to the least-significant bits, although the SPI transmits the 'word size' length in most-significant bits of the data. The function does not wait for the transmission to be completed. The user must use the SPI_chkWriteEnd function to check for its completion.

Example

```
SPI_writeWord(
  hSPI,     // handle to SPI
  SPI_DEV2, // send to device 2
  16,       // data-size: 16bits
  0xC0DE    // data-word to send
);
// ... do something ... //
while(!SPI_chkWriteEnd(hSPI)); // wait until completion
```

14.3 Register and Field Names

Table 14–2. SPI Module Register and Field Names

Register Name	Field Name(s)
SPI_SSR	L2, L1, L0, P2, P1, P0, C2, C1, C0, MSK1, MSK0, PTV
SPI_SCR	AD, NB, WR, RD
SPI_STR	WE (R), RE (R)
SPI_TXR	DATA_TX
SPI_RXR	DATA_RX

Note: R = Read only; fields not marked are Read/Write

Chapter 15

TIMER Module

The TIMER module provides APIs for interfacing to the on-chip timer devices. The 547x MCU subsystem implements three 16-bit timers configurable either in “auto-reload” or in “count-down-to-zero-and-stop” modes. Each timer can generate an interrupt to the MCU when it counts down to zero. The TIMER0 can be configured as either a watchdog counter or as a general-purpose timer. The two others (TIMER1 and TIMER2) are general-purpose timers.

Topic **Page**

15.1 Overview

Table 15–1. TIMER Descriptions

Syntax	Type	Description	Page
TIMER_Config	S	TIMER configuration structure.	15-3
TIMER_DEVICE_CNT	C	The number of timer devices.	15-3
TIMER_OPEN_RESET	C	Optional flag for TIMER_open.	15-3
TIMER_Setup	S	TIMER set-up structure.	15-3
TIMER_close	F	Closes a previously opened TIMER device.	15-4
TIMER_config	F	Configures the TIMER using configuration structure.	15-4
TIMER_getConfig	F	Reads the current TIMER configuration values.	15-5
TIMER_getCount	F	Returns the TIMER’s current count.	15-5
TIMER_getEventId	F	Obtains the event ID for the TIMER device.	15-5
TIMER_open	F	Opens TIMER device for use.	15-6
TIMER_reset	F	Resets the TIMER.	15-6
TIMER_resetAll	F	Resets all TIMER devices.	15-7
TIMER_setup	F	Configures the TIMER using the set-up structure.	15-7
TIMER_start	F	Starts the TIMER device.	15-8
TIMER_stop	F	Stops the TIMER device .	15-8

Note: C = Constant; F = Function; S = Structure

15.2 API Reference

TIMER_Config — *TIMER configuration structure*

Structure TIMER_Config

Members

Uint32 tcr	Timer control register
Uint32 cvr	Current value register

Description This is the TIMER configuration structure used to set up a TIMER device. You create and initialize this structure and then pass its address to the TIMER_config function.

TIMER_DEVICE_CNT — *The number of TIMER devices*

Constant TIMER_DEVICE_CNT

TIMER_OPEN_RESET — *Optional flag for TIMER_open*

Constant TIMER_OPEN_RESET

TIMER_Setup — *TIMER set-up structure*

Structure TIMER_Setup

Members

Uint16 period	TIMER period value (0 to 0xFFFF)
Uint16 prescale	Prescale clock TIMER value
Uint16 loadMode	Auto-reload or one-shot mode

Description This is the TIMER configuration structure used to set up a TIMER device. You create and initialize this structure and then pass its address to the TIMER_config function.

TIMER_close

Closes previously opened TIMER device

Function

```
void  TIMER_close(
    TIMER_Handle      hTimer
)
```

Arguments

hTimer   Device handle; see TIMER_open

Return Value

none

Description

Closes a previously opened TIMER device (see TIMER_open).

Example

```
TIMER_close(hTimer);
```

TIMER_config

Configures TIMER using configuration structure

Function

```
void  TIMER_config(
    TIMER_Handle      hTimer,
    TIMER_Config      *myConfig|
)
```

Arguments

hTimer   Device handle; see TIMER_open

myConfig   Pointer to the configuration structure

Return Value

none

Description

Sets up the TIMER device using the configuration structure. The values of the structure are written to the TIMER registers.

Example

```
TIMER_Config MyConfig = {
  0x001FFFE0u,  // tcr
  0x0000FFFFu   // cvr
};
...
TIMER_config(hTimer, &MyConfig);
```

TIMER_getConfig *Reads the current TIMER configuration values*

Function

```
void TIMER_getConfig(
    TIMER_Handle    hTimer,
    TIMER_Config    *config
)
```

Arguments

hTimer	Device handle; see TIMER_open
config	Pointer to the destination configuration structure

Return Value none

Description Gets the current TIMER configuration values.

Example

```
TIMER_Config TIMERCfg;
TIMER_getConfig(hTimer, &TIMERCfg);
```

TIMER_getCount *Returns TIMER's current count*

Function

```
Uint32 TIMER_getCount(
    TIMER_Handle    hTimer
)
```

Arguments hTimer Device handle; see TIMER_open

Return Value Uint32

Description This function returns the TIMER's current counter value.

Example

```
Uint32 count = TIMER_getCount(hTimer);
```

TIMER_getEventId *Obtains the event ID for the TIMER device*

Function

```
Uint16 TIMER_getEventId(
    TIMER_Handle    hTimer
)
```

Arguments hTimer Device handle; see TIMER_open

Return Value Uint16 IRQ event ID for the TIMER device

Description Use this function to obtain the event ID for the TIMER device.

Example

```
TimerEventID = TIMER_getEventId(hTimer);
IRQ_enable(TimerEventID);
```

TIMER_open — *Opens TIMER device for use*

Function

```
TIMER_Handle  TIMER_open(
    Uint16        devNum,
    Uint16        flags
)
```

Arguments

devNum — Device number:
- TIMER_DEVANY
- TIMER_DEV0
- TIMER_DEV1
- TIMER_DEV2

flags — Open flags:
- TIMER_OPEN_RESET – resets the TIMER

Return Value

TIMER_Handle — Device handle
- TIMER_INV – open failed

Description

Before a TIMER can be used, it must be ‘opened’ using this function. Once opened it cannot be opened again until it is ‘closed’ (see TIMER_close). The return value is a unique device handle that is used in subsequent TIMER API calls. If the open fails, ‘INV’ is returned.

If the OPEN_RESET flag is specified, the TIMER device registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example

```
Handle hTimer;
...
hTimer = TIMER_open(TIMER_DEV0, 0);
```

TIMER_reset — *Resets the TIMER*

Function

```
void  TIMER_reset(
    TIMER_Handle      hTimer
)
```

Arguments

hTimer — Device handle; see TIMER_open

Return Value

none

Description

Resets the TIMER device. Disables and clears the interrupt event and resets the TIMER registers to their power-on default values.

Example

```
TIMER_reset(hTimer);
```

TIMER_resetAll *Resets all TIMER devices*

Function

```
void  TIMER_resetAll(
    void
)
```

Arguments none

Return Value none

Description Resets all TIMER devices supported by the chip device by clearing and disabling the interrupt event and setting the default TIMER register values for each TIMER device (see also TIMER_reset function).

Example

```
TIMER_resetAll();
```

TIMER_setup *Configures the TIMER using the set-up structure*

Function

```
void  TIMER_setup(
    TIMER_Handle      hTimer,
    TIMER_Setup       *setup
)
```

Arguments

hTimer	Device handle; see TIMER_open
setup	The initialized set-up structure

Return Value none

Description Sets up the TIMER device using the parameters passed in via a set-up stucture. The values are written to the corresponding bit fields in the TIMER control register.

Example

```
TIMER_Setup mySetup = {
  0xFFFF,
  TIMER_PRESCALE_X4,
  TIMER_MODE_ONESHOT
}
TIMER_setup(hTimer, &mySetup);
```

TIMER_start — *Starts the TIMER device*

Function

```
void TIMER_start(
    TIMER_Handle      hTimer
)
```

Arguments hTimer — Device handle; see TIMER_open

Return Value none

Description Starts the TIMER device.

Example

```
TIMER_start(hTimer);
```

TIMER_stop — *Stops the TIMER device*

Function

```
void TIMER_stop(
    TIMER_Handle      hTimer
)
```

Arguments hTimer — Device handle; see TIMER_open

Return Value none

Description Stops the TIMER device.

Example

```
TIMER_stop(hTimer);
```

15.3 Register and Field Names

Table 15–2. TIMER Module Register and Field Names

Register Name	Field Name(s)
TIMER_TCR	LOAD_TIM (W), AR, ST, PTV
TIMER_CVR	VAL (R)

Note: R = Read only; W = Write only; fields not marked are R/W

Chapter 16

UART Module

The UART module is the key component in serial communications subsystem. This module abstracts the register descriptions provided by the UART subsystem and provides APIs that can be used to send/receive data and to configure the UART.

Topic **Page**

16.1 Overview

Table 16–1. UART Descriptions

Syntax	Type	Description	Page
UART_Config	S	UART configuration structure.	16-4
UART_DEVICE_CNT	C	Number of UART devices.	16-5
UART_DISABLE	C	Disable flag.	16-5
UART_ENABLE	C	Enable flag.	16-5
UART_OPEN_RESET	C	Flag used to reset a UART device while getting a handle.	16-5
UART_SPEED	C	Speed at which the UART operates.	16-6
UART_Setup	S	UART set-up structure.	16-6
UART_autoBaud	F	Sets UART to autobaud.	16-8
UART_close	F	Closes UART device.	16-9
UART_config	F	Sets UART configuration parameters.	16-9
UART_eventDisable	F	Disables UART interrupts.	16-10
UART_eventEnable	F	Enables UART interrupts.	16-11
UART_fget	F	Reads a character.	16-12
UART_fgets	F	Reads a string of characters.	16-12
UART_fput	F	Writes a character.	16-13
UART_fputs	F	Write a string of characters.	16-13
UART_getBaudRate	F	Gets the baud rate.	16-14
UART_getConfig	F	Gets the UART configuration parameters.	16-14
UART_getEventId	F	Gets the IRQ event of UART device.	16-14
UART_getIntType	F	Gets the type of interrupt occurred.	16-15
UART_getSetup	F	Gets the initial setup values.	16-15
UART_loopBack	F	Sets the UART device in loopback mode.	16-15
UART_open	F	Opens the UART device.	16-16
UART_read	F	Reads a buffer of characters.	16-16
UART_reset	F	Resets the UART device.	16-17

Note: C = Constant; F = Function; S = Structure

Table 16–1. UART Descriptions (Continued)

Syntax	Type	Description	Page
UART_setBaudRate	F	Sets the baudrate.	16-17
UART_setup	F	Sets up the initial parameters.	16-18
UART_write	F	Writes a buffer of characters.	16-19

Note: C = Constant; F = Function; S = Structure

16.2 API Reference

UART_Config

UART configuration structure

Structure UART_Config

Members

Uint32 fcr	FIFO control register
Uint32 scr	Status control register
Uint32 lcr	Line control register
Uint32 lsr	Line status register
Uint32 ssr	Supplementary status register
Uint32 mcr	Modem control register
Uint32 msr	Modem status register
Uint32 ier	Interrupt enable register
Uint32 isr	Interrupt status register
Uint32 efr	Enhanced feature register
Uint32 xon1	XON1 character register
Uint32 xon2	XON2 character register
Uint32 xoff1	XOFF1 character register
Uint32 xoff2	XOFF2 character register
Uint32 spr	Scratch-pad register
Uint32 div115k	Divisor for 115 kbauds generation
Uint32 divBR	Divisor for baud rate generation
Uint32 tcr	Transmission control register

Uint32 tlr	Trigger level register
Uint32 mdr	Mode definition register
Uint32 uasr	UART autobauding status register
Uint32 rdrx	RX FIFO read pointer register
Uint32 wrrx	RX FIFO write pointer register
Uint32 rdtx	TX FIFO read pointer register
Uint32 wrtx	TX FIFO write pointer register

UART_DEVICE_CNT *Number of UART devices*

Constant UART_DEVICE_CNT

UART_DISABLE *Disable flag*

Constant UART_DISABLE

Description Used to disable a flag.

Example

```
UART_loopback(hUArt,UART_DISABLE);
```

UART_ENABLE *Enable flag*

Constant UART_ENABLE

Description Used to enable a flag.

Example

```
UART_loopback(hUArt,UART_ENABLE);
```

UART_OPEN_RESET *Flag used to reset a UART device while getting a handle*

Constant UART_OPEN_RESET

Description Used to reset a UART device after open.

Example

```
UART_Handle hUart;
...
hUart = UART_open(UART_DEV0,UART_OPEN_RESET);
```

UART_SPEED — *Speed at which the UART operates*

Constant UART_SPEED

UART_Setup — *UART set-up structure*

Structure UART_Setup

Members

Uint32 eFifoEnabled — Enable or disable FIFO. The constants for the flags are:
- ❑ UART_ENABLE
- ❑ UART_DISABLE

Uint32 eRxfifoTrigLevelStart — Trigger level to start transmission:
- ❑ UART_TRIG00
- ❑ UART_TRIG04
- ❑ UART_TRIG08
- ❑ UART_TRIG12
- ❑ UART_TRIG16
- ❑ UART_TRIG20
- ❑ UART_TRIG24
- ❑ UART_TRIG28
- ❑ UART_TRIG32
- ❑ UART_TRIG36
- ❑ UART_TRIG40
- ❑ UART_TRIG44
- ❑ UART_TRIG48
- ❑ UART_TRIG52
- ❑ UART_TRIG56
- ❑ UART_TRIG60

Uint32 eRxfifoTrigLevelStop — Trigger level to stop transmission. Takes the same TRIG constants as trigger level to start transmission.

Uint32 eRxfifoTrigLevelInt — Trigger level to generate RHR interrupt. Takes the same TRIG constants as trigger level to start transmission.

Uint32 eTxfifoTrigLevelInt — Trigger level to generate THR interrupt. Takes the same TRIG constants as trigger level to start transmission.

Uint32 eWordLength	Wordlength 5 or 6 or 7 or 8. The constaants are: ❑ UART_WORD8 ❑ UART_WORD7 ❑ UART_WORD6 ❑ UART_WORD5
Uint32 eParityEnable	eParity enable or disable. The constants for the flags are: ❑ UART_ENABLE ❑ UART_DISABLE
Uint32 eParity	eParity even or odd: ❑ UART_PARITY_ODD ❑ UART_PARITY_EVEN
Uint32 eStopBits	Number of stop bits. The constants are: ❑ UART_STOP1 ❑ UART_STOP1_PLUS_HALF ❑ UART_STOP2
Uint32 eBaudRate	Baud rate for receiving or transmission. Some typical values provided as constants are: ❑ UART_BAUD_115200 ❑ UART_BAUD_115200 ❑ UART_BAUD_57600 ❑ UART_BAUD_38400 ❑ UART_BAUD_28800 ❑ UART_BAUD_19200 ❑ UART_BAUD_14400 ❑ UART_BAUD_9600 ❑ UART_BAUD_7200 ❑ UART_BAUD_4800 ❑ UART_BAUD_3600 ❑ UART_BAUD_2400 ❑ UART_BAUD_2000 ❑ UART_BAUD_1800 ❑ UART_BAUD_1200 ❑ UART_BAUD_600 ❑ UART_BAUD_300

Uint32 eLoopBackEnable	Enable loopback or not. The constants for the flags are: ❑ UART_ENABLE ❑ UART_DISABLE
Uint32 eFlowType	Flow type. The constants are: ❑ UART_FLOW_SW ❑ UART_FLOW_HW ❑ UART_FLOW_NONE
Uint32 swflowtype	Type of sw flow control
Uint32 eIntMask	Interrupt vector. The following flags can be ORed to produce the mask: ❑ UART_INT_RHR – RHR interrupt ❑ UART_INT_THR – THR interrupt ❑ UART_INT_LINE_STS – line status interrupt ❑ UART_INT_MODEM_STS – modem status interrupt ❑ UART_INT_XOFF – XOFF interrupt ❑ UART_INT_RTS – RTS interrupt ❑ UART_INT_CTS – CTS interrupt

UART_autoBaud *Sets UART to autobaud*

Function

```
void  UART_autoBaud(
    UART_Handle      hUart,
    Uint32           flag
)
```

Arguments

hUart	Device handle; see UART_open
flag	Flag to enable or disable, disable changes it to UART mode. The constants that can be used are: ❑ UART_ENABLE ❑ UART_DISABLE

Return Value none

Description Used to set the UART device, to detect the settings needed, and to configure UART device from the incoming stream.

Example

```
UART_autoBaud(hUart,UART_ENABLE);
```

UART_close — *Closes UART device*

Function

```
void  UART_close(
    UART_Handle      hUart
)
```

Arguments

hUart — Device handle; see UART_open

Return Value

none

Description

Closes a previously opened UART device (see UART_open). The following tasks are performed:

- The UART event is disabled and cleared.
- The UART registers are set to their default values.

Example

```
UART_close(hUart);
```

UART_config — *Sets UART configuration parameters*

Function

```
void  UART_config(
    UART_Handle      hUart,
    UART_Config      *config
)
```

Arguments

hUart — Device handle; see UART_open

config — Pointer to the configuration structure

Return Value

none

Description

This is the UART configuration structure used to set up a UART device. You create and initialize this structure and then pass its address to the UART_config function.

Example

```
UART_Config MyConfig = {
  0x00000051u,          // fcr
  0x00000041u,          // scr
  0x00000003u,          // lcr
  0x00000000u,          // lsr
  0x00000000u,          // ssr
  0x00000040u,          // mcr
  0x00000000u,          // msr
  0x00000000u,          // ier
  0x00000000u,          // isr
  0x00000050u,          // efr
  0x00000000u,          // xon1
  0x00000000u,          // xon2
  0x00000000u,          // xoff1
  0x00000000u,          // xoff2
  0x00000000u,          // spr
  0x000001B2u,          // div115k
  0x00000001u,          // divBR
  0x00000080u,          // tcr
  0x00000000u,          // tlr
  0x00000000u,          // mdr
  0x00000000u,          // uasr
  0x00000000u,          // rdrx
  0x00000000u,          // wrrx
  0x00000000u,          // rdtx
  0x00000000u,          // wrtx
};
...
UART_config(hUart, &MyConfig);
```

UART_eventDisable *Disables UART interrupts*

Function

```
void  UART_eventDisable(
    UART_Handle     hUart,
    Uint32          eMask
)
```

Arguments

hUart Device handle; see UART_open

	eMask	Mask specifies the events for which the interrupt is to be disabled. The following flags can be ORed to produce the mask: ❑ UART_INT_RHR – RHR interrupt ❑ UART_INT_THR – THR interrupt ❑ UART_INT_LINE_STS – line status interrupt ❑ UART_INT_MODEM_STS – modem status interrupt ❑ UART_INT_XOFF – XOFF interrupt ❑ UART_INT_RTS – RTS interrupt ❑ UART_INT_CTS – CTS interrupt

Return Value none

Description Used to disable the corresponding types of interrupts of UART device.

Example

```
UART_eventDisable(hUart, UART_INT_RHR | UART_INT_THR);
```

UART_eventEnable *Enables UART interrupts*

Function

```
void  UART_eventEnable(
    UART_Handle       hUart,
    Uint32            eMask
)
```

Arguments

hUart	Device handle; see UART_open
eMask	Mask specifies the events for which the interrupt is to be enabled. The following flags can be ORed to produce the mask: ❑ UART_INT_RHR – RHR interrupt ❑ UART_INT_THR – THR interrupt ❑ UART_INT_LINE_STS – line status interrupt ❑ UART_INT_MODEM_STS – modem status interrupt ❑ UART_INT_XOFF – XOFF interrupt ❑ UART_INT_RTS – RTS interrupt ❑ UART_INT_CTS – CTS interrupt

Return Value none

Description Used to enable the corresponding types of interrupts of UART device.

Example

```
UART_eventEnable(hUart, UART_INT_RHR | UART_INT_THR);
```

UART_fget — *Reads a character*

Function

```
Int32  UART_fget(
    UART_Handle       hUart,
    char              *c
)
```

Arguments

hUart   Device handle; see UART_open,

c   Read character

Return Value

Int32

Description

Used to read one character from the UART device. Returns 1 if character is present in the incoming stream or –1 if there is no input available.

Example

```
char c;
flag = UART_fget(hUart,&c);
if(flag) printf("read char %d",c);
```

UART_fgets — *Reads a string of characters*

Function

```
Int32  UART_fgets(
    UART_Handle       hUart,
    char              *buf,
    Uint32            nBytes
)
```

Arguments

hUart   Device handle; see UART_open

buf   Character buffer

nBytes   Buffer length

Return Value

Int32

Description

Used to read a string of characters from the UART device. Returns the character string appended with \0 or just null string (\0) if no data is present at the input.

Example

```
char buf[30];
noofchar = UART_fgets(hUart,buf,30);
while(buf[i]!='\0')
    printf("read char %d",buf[i++]);
```

UART_fput — *Writes a character*

Function

```
Int32  UART_fput(
    UART_Handle      hUart,
    char             writechar
)
```

Arguments

hUart	Device handle; see UART_open
writechar	Character to be written to output

Return Value Int32

Description Used to read one character from the UART device. Returns 1 if character can be written to the output stream or –1 if character cannot be written.

Example

```
char c=32;
flag = UART_fput(hUart,c);
```

UART_fputs — *Writes a string of characters*

Function

```
Int32  UART_fputs(
    UART_Handle      hUart,
    char             *buf
)
```

Arguments

hUart	Device handle; see UART_open
buf	Character buffer

Return Value Int32

Description Used to write a string of characters to the UART device. Returns the number of characters written.

Example

```
char buf[30];
noofchar = UART_fgets(hUart,buf);
```

UART_getBaudRate *Gets the baud rate*

Function

```
Uint32  UART_getBaudRate(
    UART_Handle      hUart
)
```

Arguments hUart Device handle; see UART_open

Return Value Uint32

Description Used to get the baud rate at which the UART device is operating.

Example

```
int baud;
baud = UART_getBaudRate(hUart);
```

UART_getConfig *Gets the UART configuration parameters*

Function

```
void  UART_getConfig(
    UART_Handle      hUart,
    UART_Config      *config
)
```

Arguments

hUart	Device handle; see UART_open
config	Pointer to the destination configuration structure

Return Value none

Description Gets configuration structure for the given UART device, which is already opened. The return structure can be modified and passed to UART_config function, if changes are to be made.

Example

```
UART_Config MyConfig;
UART_getConfig(hUart, &MyConfig);
```

UART_getEventId *Gets the IRQ event of UART device*

Function

```
Uint32  UART_getEventId(
    UART_Handle      hUart
)
```

Arguments hUart Device handle; see UART_open

Return Value Uint32

Description Use this function to obtain the event ID for the UART device.

Example

```
UartEventID = UART_getEventId(hUart);
IRQ_enable(UartEventID);
```

UART_getIntType — *Gets the type of interrupt occurred*

Function

```
Uint32  UART_getIntType(
    UART_Handle       hUart
)
```

Arguments

hUart   Device handle; see UART_open

Return Value

Uint32

Description

Used to get the type of interrupt pending with the UART device.

Example

```
intype = UART_getIntType(hUart);
```

UART_getSetup — *Gets the initial setup values*

Function

```
void  UART_getSetup(
    UART_Handle       hUart,
    UART_Setup        *setup
)
```

Arguments

hUart   Device handle

setup   Initilaization structure

Return Value

none

Description

Gets initialization structure for the given UART device, which is already opened. The return structure can be modified and passed to UART_setup function, if changes are to be made.

Example

```
UART_Setup Mysetup;
UART_getSetup(hUart,&Mysetup);
```

UART_loopBack — *Sets the UART device in loopback mode*

Function

```
void  UART_loopBack(
    UART_Handle       hUart,
    Uint32            flag
)
```

Arguments

hUart   Device handle; see UART_open

flag   flag

Return Value

none

Description

Sets the UART device in loopback mode, used for testing the UART device.

Example

```
UART_loopBack(hUart,UART_ENABLE);
```

UART_open — *Opens UART device*

Function

```
UART_Handle  UART_open(
    Uint16        devNum,
    Uint16        flags
)
```

Arguments

devNum	Specifies the device to be opened
flags	Open flags UART_OPEN_RESET – resets the UART

Return Value

UART_Handle	Device handle INV – open failed

Description

Before UART can be used, it must be 'opened' using this function. Once opened it cannot be opened again until it is 'closed' (see UART_close). The return value is a unique device handle that is used in subsequent UART API calls. If the open fails, 'INV' is returned.

If the OPEN_RESET flag is specified, the UART device registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example

```
UART_Handle hUart;
hUart = UART_open(UART_DEV0, UART_OPEN_RESET);
```

UART_read — *Reads a buffer of characters*

Function

```
Int32  UART_read(
    UART_Handle      hUart,
    char             *buf,
    Uint32           nBytes
)
```

Arguments

hUart	Device handle; see UART_open
buf	Buffer
nBytes	Number of bytes

Return Value

Int32

Description Used to read a buffer of characters from the UART device. Returns number of characters read from the incoming stream.

Example

```
char buf[30];
noofchar = UART_read(hUart,buf,30);
for(i=0;i < noofchar;i++)
    printf("read char %d",buf[i]);
```

UART_reset *Resets the UART device*

Function

```
void  UART_reset(
    UART_Handle       hUart
)
```

Arguments

hUart Device handle; see UART_open

Return Value none

Description Resets the UART device. Disables and clears the interrupt event and sets the UART registers to their power-on default values.

Example

```
UART_reset(hUart);
```

UART_setBaudRate *Sets the baud rate*

Function

```
void  UART_setBaudRate(
    UART_Handle       hUart,
    Uint32            eBaudRate
)
```

Arguments

hUart Device handle; see UART_open

eBaudRate Baud rate to be set. Baud rate can be set to any value, some typical values provided as constants are:

- UART_BAUD_115200
- UART_BAUD_115200
- UART_BAUD_57600
- UART_BAUD_38400
- UART_BAUD_28800
- UART_BAUD_19200
- UART_BAUD_14400
- UART_BAUD_9600

- UART_BAUD_7200
- UART_BAUD_4800
- UART_BAUD_3600
- UART_BAUD_2400
- UART_BAUD_2000
- UART_BAUD_1800
- UART_BAUD_1200
- UART_BAUD_600
- UART_BAUD_300

Return Value none

Description Used to set the baud rate at which the UART device should operate.

Example

```
UART_setBaudRate(hUart,UART_BAUD_115200);
```

UART_setup *Sets up initial parameters*

Function

```
void  UART_setup(
    UART_Handle       hUart,
    UART_Setup        *setup
)
```

Arguments

hUart	Device handle
setup	Initilaization structure

Return Value none

Description This is the UART initialization structure used to set up a UART device. You can create and initialize this structure and then pass its address to the UART_setup function.

Example

```
UART_setup mySetup{
UART_ENABLE,        // fifo enabled
UART_TRIG00,        // rx fifo trigger level, start transmissions
UART_TRIG08,        // rx fifo trigger level, stop transmissions
UART_TRIG08,        // rx fifo trigger levels, generate interrupts
UART_TRIG08,        // tx fifo trigger levels, generate interrupts
UART_WORD8,         // word length
UART_ENABLE,        // eParity enable
UART_PARITY_ODD,    // eParity type
UART_STOP1,         // number of stop bits
UART_BAUD_115200,   // baud rate
UART_DISABLE,       // loopback enable
UART_FLOW_HW,       // flow type
0,                  // type of sw flow
UART_INT_RHR|UART_INT_THR,  // interrupt mask
};
UART_setup(hUart,&mySetup);
```

UART_write — *Writes a buffer of characters*

Function

```
Int32  UART_write(
    UART_Handle     hUart,
    char            *buf,
    Uint32          nBytes
)
```

Arguments

hUart	Device handle; see UART_open
buf	Buffer
nBytes	Number of bytes

Return Value Int32

Description Used to write a buffer of characters to the UART device. Returns number of character written to the outgoing stream.

Example

```
char buf[30]={20,30,....};
noofchar = UART_write(hUart,buf,30);
```

16.3 Register and Field Names

Table 16–2. UART Module Register and Field Names

Register Name	Field Name(s)
UART_RHR	RX_BI (R), RX_FE (R), RX_PE (R), RHR (R)
UART_THR	THR (W)
UART_FCR	RXF_TR, TXF_TR, RXF_CL, TXF_CL, FIFO_EN
UART_SCR	FINIT_ST, FINIT, RXCTUP_EN, TX_E_CTL_IT, FPTRACEN
UART_LCR	BREAK_EN, PAR_T2, PAR_T1, PAR_EN, NB_STOP, C_LN
UART_LSR	RX_FIFO_STS, TX_SR_E, TX_FIFO_E, RX_OE, RX_FIFO_E
UART_SSR	RX_CT_WUP_STS, TX_FIFO_FULL
UART_MCR	CLKSEL, TCR_TLR, XON_EN, LOOPBACK_EN, RTS, DCD
UART_MSR	NDSR_STS, NCTS_STS, DSR_STS, CTS_STS
UART_IER	CTS_IT, RTS_IT, XOFF_IT, M_IT, L_IT, TH_IT, RH_IT
UART_ISR	FCR_M1, FCR_M2, IT_TYPE, IT_PENDING
UART_EFR	ACTSEN, ARTSEN, SP_CHAR, ENHANCED_EN, SW_FLOW
UART_XON1	XON_WORD1
UART_XON2	XON_WORD2
UART_XOFF1	XOFF_WORD1
UART_XOFF2	XOFF_WORD2
UART_SPR	SPR_WORD
UART_DIV115K	DIV_115K
UART_DIVBR	DIV_BITRATE
UART_TCR	RXF_TR_START, RXF_TR_HALT
UART_TLR	RXF_TR_RHR, TXF_TR_THR

Note: R = Read only; W = Write only; fields not marked are R/W

Table 16–2. UART Module Register and Field Names (Continued)

Register Name	Field Name(s)
UART_MDR	MODE_SELECT
UART_UASR	PAR_T, BIT_BY_CHAR, STOP_BIT, SPEED
UART_RDRX	RX_READ_PTR
UART_WRRX	RX_WRITE_PTR
UART_RDTX	TX_READ_PTR
UART_WRTX	TX_WRITE_PTR

Note: R = Read only; W = Write only; fields not marked are R/W

Chapter 17

WDTIM Module

The WDTIM module provides APIs for interfacing to the on-chip watchdog timer device. By default, the first timer device (TIMER0) is configured as a watchdog timer. The watchdog is designed to detect user programs stuck in infinite loops resulting in loss of program control or "runaway" programs. The watchdog timer will reset the ARM MCU subsystem if it counts down to zero, hence the user program should periodically reload the watchdog.

Topic **Page**

17.1 Overview

Table 17–1. WDTIM Descriptions

Syntax	Type	Description	Page
WDTIM_Config	S	WDTIM configuration structure.	17-3
WDTIM_DEVICE_CNT	C	The number of watchdog timers.	17-3
WDTIM_OPEN_RESET	C	Optional flag for WDTIM_open.	17-3
WDTIM_close	F	Closes previously opened watchdog timer.	17-3
WDTIM_config	F	Configures the watchdog timer using a configuration structure.	17-4
WDTIM_getConfig	F	Reads the current WDTIM configuration values.	17-4
WDTIM_open	F	Opens the watchdog timer for use.	17-5
WDTIM_reset	F	Resets the watchdog timer.	17-5
WDTIM_service	F	Services (kicks) the watchdog timer.	17-6
WDTIM_start	F	Starts the watchdog timer.	17-6

Note: C = Constant; F = Function; S = Structure

17.2 API Reference

WDTIM_Config — *WDTIM configuration structure*

Structure WDTIM_Config

Members

Uint32 tcr	Timer control register
Uint32 cvr	Current value register

Description This is the watchdog configuration structure used to set up the watchdog timer device. You create and initialize this structure and then pass its address to the WDTIM_config function.

WDTIM_DEVICE_CNT — *The number of watchdog timers*

Constant WDTIM_DEVICE_CNT

WDTIM_OPEN_RESET — *Optional flag for WDTIM_open*

Constant WDTIM_OPEN_RESET

WDTIM_close — *Closes previously opened watchdog timer*

Function

```
void  WDTIM_close(
    WDTIM_Handle      hWDTim
)
```

Arguments

hWDTim	Device handle; see WDTIM_open

Return Value none

Description Closes a previously opened watchdog device (see WDTIM_open). The following tasks are performed:

- The watchdog functionality is disabled.
- The timer registers are set to their default values.

Example

```
WDTIM_close(hWDTim);
```

WDTIM_config — *Configures watchdog timer using a configuration structure*

Function

```
void  WDTIM_config(
    WDTIM_Handle      hWDTim,
    WDTIM_Config      *config
)
```

Arguments

hWDTim	Device handle; see WDTIM_open
config	Pointer to the configuration structure

Return Value none

Description Sets up the timer device using the configuration structure. The values of the structure are written to the TIMER registers.

Example

```
WDTIM_Config myConfig = {
  0x001FFFE0u,  // tcr
  0x0000FFFFu   // cvr
};
...
WDTIM_config(hWDTim, &myConfig);
```

WDTIM_getConfig — *Reads the current WDTIM configuration values*

Function

```
void  WDTIM_getConfig(
    WDTIM_Handle      hWDTim,
    WDTIM_Config      *config
)
```

Arguments

hWDTim	Device handle; see WDTIM_open
config	Pointer to the destination configuration structure

Return Value none

Description Gets the current timer configuration values.

Example

```
WDTIM_Config wdtimCfg;
WDTIM_getConfig(hWDTim, &wdtimCfg);
```

WDTIM_open *Opens watchdog timer for use*

Function

```
WDTIM_Handle  WDTIM_open(
    Uint16         devNum,
    Uint16         flags
)
```

Arguments

devNum — Device number:
- WDTIM_DEVANY
- WDTIM_DEVICE

flags — Open flags:
- WDTIM_OPEN_RESET – resets the watchdog

Return Value

WDTIM_Handle — Device handle:
- INV – open failed

Description

Before the watchdog timer can be used, it must be ‘opened’ using this function. Once opened it cannot be opened again until it is ‘closed’ (see WDTIM_close). Since there is just one watchdog timer, all APIs in this module (except for WDTIM_close) ignore the handle value. If the open fails, it returns ‘INV’.

If the WDTIM_OPEN_RESET flag is specified, the watchdog timer device registers are set to their power-on defaults.

Example

```
WDTIM_Handle hWDT;
...
hWDT = WDTIM_open(WDTIM_DEV0, 0);
```

WDTIM_reset *Resets the watchdog timer*

Function

```
void  WDTIM_reset(
    WDTIM_Handle      hWDTim
)
```

Arguments

hWDTim — Device handle; see WDTIM_open

Return Value

none

Description

Resets the watchdog timer device. The function disables the watchdog functionality and then sets the timer registers to their power-on default values.

Example

```
WDTIM_reset(hWDTim);
```

WDTIM_service — *Services (kicks) the watchdog timer*

Function

```
void WDTIM_service(
    WDTIM_Handle    hWDTim,
    Uint16          period
)
```

Arguments

hWDTim	Device handle; see WDTIM_open
period	The watchdog timer period

Return Value none

Description The WDTIM_service() function must be called periodically to prevent a watch dog timeout. Once the watchdog timer is started, the user's program must reload the watchdog timer before the counter underflows by calling this function. A counter underflow resets the ARM core and all modules controlled by it.

Example

```
WDTIM_service(0, 0xffff);
```

WDTIM_start — *Starts the watchdog timer*

Function

```
void WDTIM_start(
    WDTIM_Handle    hWDTim
)
```

Arguments

hWDTim	Device handle; see WDTIM_open

Return Value none

Description Starts the watchdog timer device. Once the watchdog is started, the user's program must reload the watchdog timer before the counter underflows by calling the WDTIM_service function.

Example

```
WDTIM_start(hWDTim);
```

17.3 Register and Field Names

Table 17–2. WDTIM Module Register and Field Names

Register Name	Field Name(s)
WDTIM_TCR	WDS, TM, LOAD_TIM (W), ST
WDTIM_CVR	VAL (R)

Note: R = Read only; W = Write only; fields not marked are R/W