Freescale MQX™ I/O Drivers User Guide

MQXIOUG
Rev. 18
04/2013

freescale™

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: MQXIOUG
Rev. 18, 04/2013

freescale™

Chapter 1
Before You Begin

Chapter 2
MQX I/O

Chapter 3
Null-Device Driver

Chapter 4
Pipe Device Driver

Chapter 5
Serial-Device Families

Chapter 6
Simple Memory Driver

Chapter 7
GPIO Driver

Chapter 8
ADC Driver

Chapter 9
SPI Drivers

Chapter 10
Legacy SPI Drivers

Chapter 11
I2C Driver

Chapter 12
FlashX Driver

Chapter 13
SD Card Driver

Chapter 14
RTC Driver

Chapter 15
ESDHC Driver

Chapter 16
FlexCAN Driver

Chapter 17
NAND Flash Driver

Chapter 18
DAC Driver

Chapter 19
LWGPIO Driver

Chapter 20
Low Power Manager

Chapter 21
Resistive Touch Screen Driver

Chapter 22
LWADC Driver

Chapter 23
Debug IO Driver

Chapter 24
HMI

Chapter 25
I2S Driver

Chapter 26
HWTIMER Driver

Revision History

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to http://www.freescale.com/mqx.

The following revision history table summarizes changes contained in this document.

Revision Number	Revision Date	Description of Changes
Rev. 0	03/2009	Initial Release coming with MQX 3.1
Rev. 1	05/2009	Update done for MQX 3.2. GPIO, ADC, SPI and FlashX driver description added.
Rev. 2	05/2009	Update done for MQX 3.3. I2C driver description added.
Rev. 3	09/2009	Update done for MQX 3.4. SD Card driver description added. New SPI commands described. More detailed FlashX example added.
Rev. 4	01/2010	Updated for MQX 3.5. RTC driver description added. New SPI, ADC and GPIO commands described. New FlashX commands for dual-internal flash devices described.
Rev. 5	05/2010	SPI, I2C, ADC and RTC sections updated. io_open -> io_fopen io_close -> io_fclose Added the following chapters: ESDHC Driver FlexCAN Driver DAC Driver NAND Flash Driver Updated SD Card Driver chapter
Rev. 6	08/2010	IO_IOCTL_SPI_KEEP_QSPI_CS_ACTIVE SPI driver IOCTL command description added.
Rev. 7	11/2010	Description of IO_SERIAL_NON_BLOCKING serial driver open flag added. The following chapters were updated: RTC Driver NAND Flash Driver
Rev. 8	02/2011	The Serial-Device Families and NAND Flash Driver chapters were updated. LWGPIO Driver chapter added.
Rev. 9	04/2011	LWGPIO Driver, ADC Driver and FlashX Driver chapters were updated.

Revision Number	Revision Date	Description of Changes
Rev. 10	12/2011	Low Power Manager chapter added. Resistive Touch Screen Driver chapter added. Debug IO Driver chapter added. IO_IOCTL_SERIAL_START_BREAK and IO_IOCTL_SERIAL_STOP_BREAK description removed as it is not implemented yet. FlashX Driver and ADC Driver chapters updated. "Initialization Record" sections of the following chapters updated: Serial-Device Families SD Card Driver SPI Driver I2C Driver ADC Driver ESDHC Driver
Rev. 11	03/2012	LWADC Driver chapter added.
Rev. 12	04/2012	Low Power Manager chapter updated.
Rev. 13	05/2012	Debug IO Driver chapter added.
Rev. 14	06/2012	HMI chapter added. Description of serial driver I/O Control Commands updated.
Rev. 15	10/2012	ESDHC chapter updated. I2S Driver chapter added. lwgpio_toggle_pin_output function description updated.
Rev. 16	11/2012	FLASHX_INIT_STRUCT description corrected. SPI Driver section split into 2 sections: one for the Legacy SPI driver and the second one for the updated SPI driver.
Rev. 17	02/2013	"Using the Driver" section in "SPI Drivers" chapter updated. HWTIMER Driver chapter added.
Rev. 18	04/2013	Description of I/O Subsystem interface functions added into the I/O Subsystem section. API Function Reference of the LWGPIO Driver updated.

Chapter 1 Before You Begin

1.1 About This Book

MQX includes a large number of I/O device drivers, which we group into driver families according to the I/O device family that they support. Each driver family includes a number of drivers, each of which supports a particular device from its device family.

Use this book in conjunction with:

- *MQX Users Guide*
- *MQX API Reference Manual*
- Driver source code

Use this book in conjunction with *MQX Users Guide*, which covers the following general topics:

- MQX at a glance
- Using MQX
- Rebuilding MQX
- Developing a new BSP
- Frequently asked questions
- Glossary of terms

1.2 About MQX

MQX is a real-time operating system from MQX Embedded and ARC. It has been designed for uniprocessor, multiprocessor, and distributed-processor embedded real-time systems.

To leverage the success of the MQX RTOS, Freescale Semiconductor adopted this software platform for ColdFire® and Power Architecture® families of microprocessors. Comparing to the original MQX distributions, the Freescale MQX distribution is simpler to configure and use. One single release now contains the MQX operating system in addition to all the other software components supported for a given microprocessor part. The first MQX version released as Freescale MQX RTOS is assigned a number 3.0. It is based on and is API-level compatible with the MQX RTOS version 2.50 released by ARC.

MQX RTOS is a runtime library of functions which programs use to become real-time multitasking applications. The main features are its scalable size, component-oriented architecture, and ease of use.

MQX RTOS supports multiprocessor applications and can be used with flexible embedded I/O products for networking, data communications, and file management.

In this document, MQX stands for MQX Real Time Operating System.

1.3 Document Conventions

1.3.1 Notes

Notes point out important information. For example:

NOTE

Non-strict semaphores do not have priority inheritance.

1.3.2 Cautions

Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware. For example:

CAUTION

If you modify MQX data types, some MQX host tools may not operate properly.

Chapter 2 MQX I/O

2.1 Overview

This section describes how I/O device drivers fit into the MQX I/O model. It includes the information that applies to all driver families and their members. I/O device drivers are dynamically (or in run-time) installed software packages that provide a direct interface to hardware.

2.2 MQX I/O Layers

The MQX I/O model consists of three layers of software:

- Formatted (ANSI) I/O
- MQX I/O Subsystem (Called from the Formatted I/O)
- MQX I/O Device Drivers (Called from the MQX I/O Subsystem)

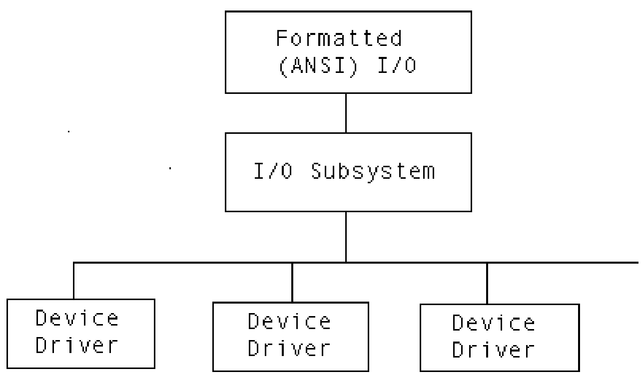


Figure 2-1. MQX I/O Layers

As a result of MQX layered approach, it is possible for device drivers to open and access other device drivers. For example, the I/O PCB device drive sends out a packet by opening and using an asynchronous character device driver.

2.2.1 I/O Device Structure

Figure below shows the relationship between a file handle (FILE_STRUCT) that is returned by **fopen()**, the I/O device structure (allocated when the device is installed), and I/O driver functions for all I/O device drivers.

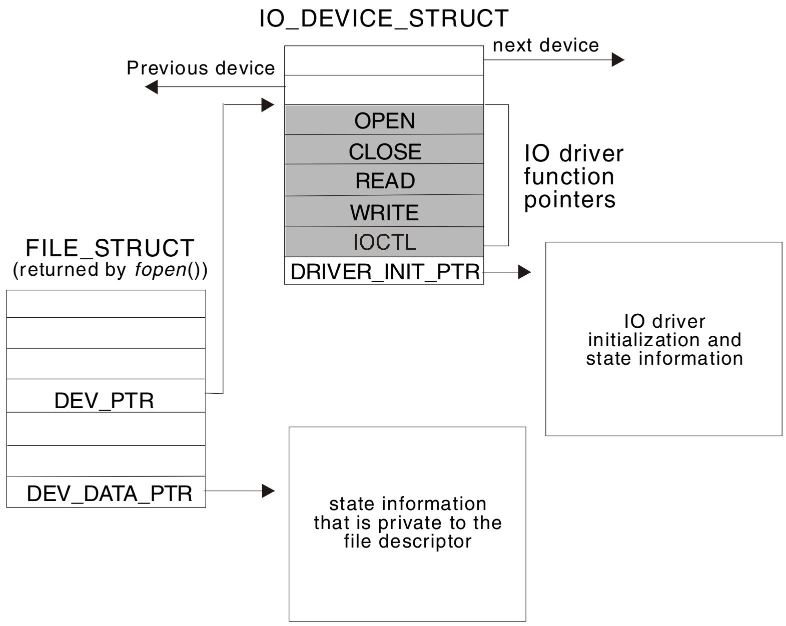


Figure 2-2. I/O Device Structure — I/O Device Drivers

2.2.2 I/O Device Structure for Serial-Device Drivers

Serial device drivers are complex in that they have a generic driver layer and a low-level standard simple interface to the serial hardware.

Figure below shows the relationship between a file handle (FILE_STRUCT) that is returned by **fopen()**, the I/O device structure (allocated when the device is installed), and upper-level serial-device driver functions.

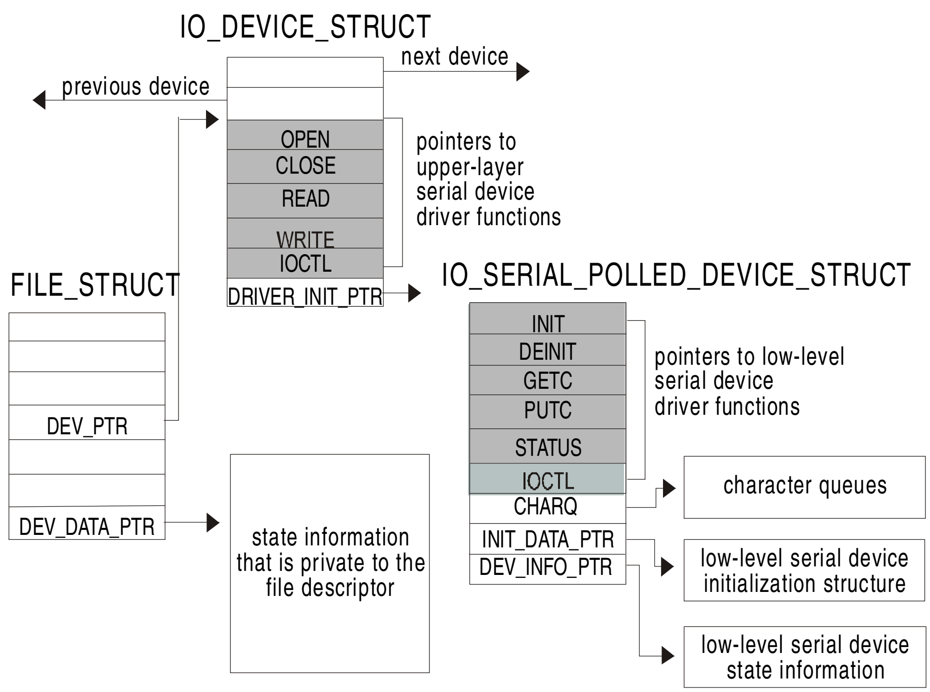


Figure 2-3. I/O Device Structure — Serial-Device Drivers

2.3 Formatted I/O Library

The MQX formatted I/O library is a subset implementation of the ANSI C standard library. The library makes calls to the I/O subsystem.

To use the formatted I/O library, include the header file *fio.h*. This file also contains ANSI-like aliases to official MQX API calls:

ANSI C call	MQX API
clearerr	_io_clearerr
fclose	_io_fclose
feof	_io_feof
ferror	_io_ferror
fflush	_io_fflush

ANSI C call	MQX API
fgetc	_io_fgetc
fgetline	_io_fgetline
fgets	_io_fgets
fopen	_io_fopen
fprintf	_io_fprintf
fputc	_io_fputc
fputs	_io_fputs
fscanf	_io_fscanf
fseek	_io_fseek
fstatus	_io_fstatus
ftell	_io_ftell
fungetc	_io_fungetc
ioctl	_io_ioctl
printf	_io_printf
putc	_io_fputc
read	_io_read
scanf	_io_scanf
sprintf	_io_sprintf
sscanf	_io_sscanf
vprintf	_io_vprintf
vfprintf	_io_vfprintf
vsprintf	_io_vsprintf
write	_io_write

2.4 I/O Subsystem

The MQX I/O subsystem implementation is a slightly deviated subset of the POSIX standard I/O.
It follows the UNIX model of **open**, **close**, **read**, **write**, and **ioctl** functions. The I/O subsystem makes calls to I/O device-driver functions. MQX I/O uses pointers to FILE as returned by **fopen()**, instead of file descriptors (FDs).

The following functions can be used to interface the I/O Subsystem:

- _io_dev_install
- _io_dev_install_ext
- _io_dev_uninstall
- _io_get_handle

- _io_init
- _io_set_handle

2.4.1 _io_dev_install

This function installs a device dynamically, so tasks can fopen to it.

Synopsis

```
_mqx_uint _io_dev_install(
    char_ptr      identifier,
    IO_OPEN_FPTR  io_open,
    IO_CLOSE_FPTR io_close,
    IO_READ_FPTR  io_read,
    IO_WRITE_FPTR io_write,
    IO_IOCTL_FPTR io_ioctl,
    pointer       io_init_data_ptr);
```

Parameters

- *identifier [IN]* — A string that identifies the device for fopen.
- *io_open [IN]* — The I/O open function.
- *io_close [IN]* — The I/O close function.
- *io_read [IN]* — The I/O read function.
- *io_write [IN]* — The I/O write function.
- *io_ioctl [IN]* — The I/O ioctl function.
- *io_init_data_ptr [IN]* — The I/O initialization data.

Return Value

- MQX_OK (success)
- MQX_INVALID_PARAMETER (failure: a NULL pointer provided or none delimiter found in the identifier string or more than 1 delimiter found in the identifier string or the identifier was composed of a single delimiter only)
- IO_DEVICE_EXISTS (failure: device already installed)
- MQX_OUT_OF_MEMORY(failure: MQX cannot allocate memory for the device)

2.4.2 _io_dev_install_ext

This function installs a device dynamically, so tasks can fopen to it. In comparison with **_io_dev_install** this function also registers an uninstall function.

Synopsis

```
_mqx_uint _io_dev_install(
    char_ptr      identifier,
    IO_OPEN_FPTR  io_open,
    IO_CLOSE_FPTR io_close,
    IO_READ_FPTR  io_read,
    IO_WRITE_FPTR io_write,
    IO_IOCTL_FPTR io_ioctl,
    IO_UNINSTALL_FPTR io_uninstall,
    pointer       io_init_data_ptr);
```

Parameters

- *identifier [IN]* — A string that identifies the device for fopen.
- *io_open [IN]* — The I/O open function.
- *io_close [IN]* — The I/O close function.
- *io_read [IN]* — The I/O read function.
- *io_write [IN]* — The I/O write function.
- *io_ioctl [IN]* — The I/O ioctl function.
- *io_uninstall [IN]* — The I/O un-install function.
- *io_init_data_ptr [IN]* — The I/O initialization data.

Return Value

- MQX_OK (success)
- MQX_INVALID_PARAMETER (failure: a NULL pointer provided or none delimiter found in the identifier string, or more than 1 delimiter found in the identifier string or the identifier was composed of a single delimiter only)
- IO_DEVICE_EXISTS (failure: device already installed)
- MQX_OUT_OF_MEMORY(failure: MQX cannot allocate memory for the device)

2.4.3 _io_dev_uninstall

This function uninstalls a device dynamically.

Synopsis

```
_mqx_int _io_dev_uninstall(
    char_ptr  identifier);
```

Parameters

identifier [IN] — A string that identifies the device for fopen.

Return Value

- IO_OK (success)
- IO_DEVICE_DOES_NOT_EXIST (failure: device not installed)
- The I/O un-install function return values.

2.4.4 _io_get_handle

This function returns the address of a default standard I/O FILE. If an incorrect type is given, or the file_ptr has not been specified, the function returns NULL.

Synopsis

```
pointer _io_get_handle(
    _mqx_uint stdio_type);
```

Parameters

- *stdio_type [IN]* — Which I/O handle to return.

Return Value

- I/O handle (success)
- NULL (failure)

2.4.5 _io_init

This function initializes the kernel I/O subsystem.

Synopsis

```
_mqx_uint _io_init(void);
```

Parameters

- None

Return Value

- MQX_OK (success)
- _lwsem_create function return values

2.4.6 _io_set_handle

This function changes the address of a default I/O handle, and returns the previous one. If an incorrect type is given, or the I/O handle was uninitialized, NULL is returned.

Synopsis

```
pointer _io_set_handle(
    _mqx_uint stdio_type,
    pointer new_file_ptr);
```

Parameters

- *stdio_type [IN]* — Which I/O handle to modify.
- *new_file_ptr [IN]* — The new I/O handle.

Return Value

- Previous I/O handle or NULL.

2.5 I/O Error Codes

The general error code for all I/O functions is IO_ERROR (–1). Some driver families, their members, or both, may have error codes that are specific to them. See the chapter that describes the driver family for more details. Also, see source code of public header files implementing the driver functionality.

2.6 I/O Device Drivers

I/O device drivers provide a direct interface to hardware modules and are described in Section 2.9, "Device Driver Services" below.

2.7 Device Names

The string that identifies the name of a device must end with **:**.

For example:

```
_io_mfs_install("mfs1:" ...)
```

installs device `mfs1:`

Characters following **:** are considered as extra information for the device (passed to the device driver by **fopen()** call).

For example:

```
fopen("mfs1:bob.txt")
```

opens file *bob.txt* on device `mfs1:`

2.8 Installing Device Drivers

To install a device driver, follow either of the steps below:

- Call **_io_device_install()** (where **device** is replaced by the name of the driver family) from your application. Usually, the function calls **_io_dev_install()** internally to register the device with MQX. It also performs device-specific initialization, such as allocating scratch memory and initializing other MQX objects needed for its operation (for example semaphores).

- Call **_io_dev_install()** directly from the BSP or your application. The function registers the device with MQX.

See Section 2.7, "Device Names" above for restrictions on the string that identifies the name of a device.

2.9 Device Driver Services

A device driver usually provides the following services:

- __io_device_open
- _io_device_close
- _io_device_read
- _io_device_write
- _io_device_ioctl

2.9.1 _io_device_open

This driver function is required. By convention, the function name is composed as **_io_device_open**, where **device** is a placeholder for custom device driver name.

Synopsis

```
mqx_int _io_device_open(
        FILE_DEVICE_STRUCT_PTR fd_ptr,
        char _PTR_ open_name_ptr,
        char _PTR_ open_mode_flags);
```

Parameters

- *fd_ptr [IN]* — Pointer to a file device structure that the I/O subsystem passes to each I/O driver function.
- *open_name_ptr [IN]* — Pointer to the remaining portion of the string (after the device name is removed) used to open the device.
- *open_mode_flags [IN]* — Pointer to the open mode flags passed from **fopen()**.

Remarks

This function is called when user application opens the device file using the **fopen()** call.

Return Value

This function returns MQX_OK if successful, or an appropriate error code.

2.9.2 _io_device_close

This driver function is required. By convention, the function name is composed as **_io_device_close**, where **device** is a placeholder for custom device driver name.

Synopsis

```
mqx_int _io_device_close(
        FILE_DEVICE_STRUCT_PTR fd_ptr);
```

Parameters

- *fd_ptr [IN]* — File handle for the device being closed.

Remarks

This function is called when user application closes the device file using the **fclose()** call.

Return Value

This function returns MQX_OK if successful, or an appropriate error code.

2.9.3 _io_device_read

This driver function is optional and is implemented only if device is to provide a “read” call.
By convention, the function name is composed as **_io_device_read**, where **device** is a placeholder for custom device driver name.

Synopsis

```
mqx_int _io_device_read(
        FILE_DEVICE_STRUCT_PTR fd_ptr,
        char _PTR_ data_ptr,
        _mqx_int_ num);
```

Parameters

- *fd_ptr [IN]* — File handle for the device.
- *data_ptr [OUT]* — Where to write the data.
- *num [IN]* — Number of bytes to be read.

Return Value

This function returns the number of bytes read from the device or IO_ERROR (negative value) in case of error.

Remarks

This function is called when user application tries to read bytes from device using the **read()** call.

2.9.4 _io_device_write

This driver function is optional and is implemented only if device is to provide a “write” call.
By convention, the function name is composed as **_io_device_write**, where **device** is a placeholder for custom device driver name.

Synopsis

```
mqx_int _io_device_write(
        FILE_DEVICE_STRUCT_PTR fd_ptr,
        char _PTR_ data_ptr,
        _mqx_int_ num);
```

Parameters

- *fd_ptr [IN]* — File handle for the device.
- *data_ptr [IN]* — Where the data is.
- *num [IN]* — Number of bytes to write.

Return Value

This function returns the number of bytes written to the device or IO_ERROR (negative value) in case of error.

Remarks

This function is called when user application tries to write a block of data into device using the **write()** call.

2.9.5 _io_device_ioctl

This driver function is optional and should be implemented only if device is to provide an “ioctl” call. By convention, the function name is composed as **_io_device_ioctl**, where **device** is a placeholder for custom device driver name.

Synopsis

```
mqx_int _io_device_ioctl(
        FILE_DEVICE_STRUCT_PTR fd_ptr,
        _mqx_int_ cmd
        pointer param_ptr);
```

Parameters

- *fd_ptr [IN]* — File handle for the device.
- *cmd [IN]* — I/O control command (see Section 2.10, “I/O Control Commands”).
- *param_ptr [IN/OUT]* — Pointer to the I/O control parameters.

Return Value

This function typically returns MQX_OK in case of success, or an error code otherwise.

Remarks

This function is called when user application tries to execute device-specific control command using the **ioctl()** call.

2.10 I/O Control Commands

The following I/O control commands are standard for many driver families and are also mapped to dedicated MQX system calls. Depending on the family, all of them may or may not be implemented.

I/O control command	Description
IO_IOCTL_CHAR_AVAIL	Check for the availability of a character.
IO_IOCTL_CLEAR_STATS	Clear the driver statistics.
IO_IOCTL_DEVICE_IDENTIFY	Query a device to find out its properties (see Section 2.11, "Device identification").
IO_IOCTL_FLUSH_OUTPUT	Wait until all output has completed.
IO_IOCTL_GET_FLAGS	Get connection-specific flags.
IO_IOCTL_GET_STATS	Get the driver statistics.
IO_IOCTL_SEEK	Seek to the specified byte offset.
IO_IOCTL_SEEK_AVAIL	Check whether a device can seek.
IO_IOCTL_SET_FLAGS	Set connection-specific flags.

2.11 Device identification

When **_io_device_ioctl()** function is invoked with IO_IOCTL_DEVICE_IDENTIFY command, the *param_ptr* is the address of a three-entry array. Each entry is of type **uint_32**.

The function returns the following properties in the array:

- IO_DEV_TYPE_PHYS_*XXX* – Physical device type. For example, IO_DEV_TYPE_PHYS_SPI
- IO_DEV_TYPE_LOGICAL_XXX – Logical device type. For example, IO_DEV_TYPE_LOGICAL_MFS
- IO_DEV_ATTR_XXX – Device attributes bitmask. For example, IO_DEV_ATTR_READ

2.12 Error Codes

A success in device driver call is signalled by returning IO_OK constant which is equal to MQX_OK. An error is signalled by returning IO_ERROR. The driver writes detailed information about the error in the ERROR field of the FILE_STRUCT. You can determine the error by calling **ferror()**.

The I/O error codes for the ERROR field are as follows:

- IO_DEVICE_EXISTS
- IO_DEVICE_DOES_NOT_EXIST
- IO_ERROR_DEVICE_BUSY
- IO_ERROR_DEVICE_INVALID
- IO_ERROR_INVALID_IOCTL_CMD
- IO_ERROR_READ

- IO_ERROR_READ_ACCESS
- IO_ERROR_SEEK
- IO_ERROR_SEEK_ACCESS
- IO_ERROR_WRITE
- IO_ERROR_WRITE_ACCESS
- IO_ERROR_WRITE_PROTECTED
- IO_OK

2.13 Driver Families

MQX supports a number of driver families, some of them described in this manual. This manual includes the following information for the drivers:

- General information about the family
- I/O control functions that may be common to the family
- Error codes that may be common to the family

2.14 Families Supported

The following table lists the driver families that MQX supports. The second column is the device in the name of the I/O driver functions. For example, for serial devices operating in polled mode the **_io_device_open()** becomes **_io_serial_polled_open()**.

NOTE

The information provided in the next sections is based on original documentation accompanying the previous versions of MQX. Some of the drivers described here may not yet be supported by Freescale MQX release.

Also, not all drivers available in the Freescale MQX are documented in this document. Please reffer to *MQX Release Notes* for the list of supported drivers.

Drivers	Family (device)	Directory in mqx\source\io
DMA	`dma`	`dma`
Ethernet	`enet`	`enet`
Flash devices	`flashx`	`flashx`
Interrupt controllers	various controllers	`int_ctrl`
Non-volatile RAM	`nvram`	`nvram`
Null device (void driver)	`null`	`io_null`
PCB (Packet Control Block) drivers (HDLC, I^2C, ..)	`pcb`	`pcb`

Drivers	Family (device)	Directory in mqx\source\io
PC Card devices	pccard	pccard
PC Card flash devices	pcflash	pcflash
PCI (Peripheral Component Interconnect) devices	pci	pci
UART Serial devices: asynchronous polled, asynchronous interrupt	serial	serial
Simple memory	mem	io_mem
Timers	various controllers	timer
USB	usb	usb
Real-time clock	rtc	rtc
I^2C (non-PCB, character-wise)	i2c	i2c
QSPI (non-PCB, character-wise)	qspi	qpsi
General purpose I/O	gpio	gpio
Dial-up networking interface	dun	io_dun

NOTE

Some of the device drivers such as Timer, CAN, RTC, etc. and the interrupt controller drivers implement custom API and do not follow the standard driver interface.

NOTE

When this manual was written, Freescale MQX did not support PCB-based I^2C and QSPI drivers. Only character-based master-mode-only I^2C and QSPI drivers are supported.

Chapter 3 Null-Device Driver

3.1 Overview

The null device driver provides an I/O device that functions as a device driver but does not perform any work.

3.2 Source Code Location

Source code for the null-device driver is in *source\io\io_null*.

3.3 Header Files

To use the null-device driver, include the header file *io_null.h* in your application or in the BSP file *bsp.h*.

3.4 Driver Services

The null-device driver provides the following services:

API	Calls
_io_fopen()	_io_null_open()
_io_fclose()	_io_null_close()
_io_read()	_io_null_read()
_io_write()	_io_null_write()
_io_ioctl()	_io_null_ioctl()

3.5 Installing the Driver

The null-device driver provides an installation function that either the BSP or the application calls. The function installs the **_io_null** family of functions and calls **_io_dev_install()**.

```
_mqx_uint _io_null_install
   (
      /* [IN] A string that identifies the device for fopen */
      char_ptr            identifier
   )
```

3.6 I/O Control Commands

There are no I/O control commands for **_io_ioctl()**.

3.7 Error Codes

The null-device driver does not add any additional error codes.

Chapter 4 Pipe Device Driver

4.1 Overview

This section contains the information applicable for the pipe device driver accompanying MQX. The pipe device driver provides a blocking, buffered, character queue that can be read and written to by multiple tasks.

4.2 Source Code Location

The source code for the pipe device driver is in *source\io\pipe*.

4.3 Header Files

To use the pipe device driver, include the header file *pipe.h* in your application or in the BSP file *bsp.h*.

The file *pipe_prv.h* contains private constants and data structures that the driver uses. You must include this file if you recompile the driver. You may also want to look at the file as you debug your application.

4.4 Driver Services

The pipe device driver provides the following services:

API	Calls
_io_fopen()	_io_pipe_open()
_io_fclose()	_io_pipe_close()
_io_read()	_io_pipe_read()
_io_write()	_io_pipe_write()
_io_ioctl()	_io_pipe_ioctl()

4.5 Installing Drivers

The pipe device driver provides an installation function that either the BSP or the application calls. The function installs the **_io_pipe** family of functions and calls **_io_dev_install()**.

```
_mqx_uint _io_pipe_install
   (
      /* [IN] A string that identifies the device for fopen */
      char_ptr            identifier,
      /* [IN] The pipe queue size to use */
      uint_32             queue_size,
      /* [IN] Currently not used */
```

```
    uint_32            flags
)
```

4.6 Reading From and Writing To a Pipe

When a task calls **_io_write()**, the driver writes the specified number of bytes to the pipe. If the pipe becomes full before all the bytes are written, the task blocks until there is space available in the pipe. Space becomes available only if another task reads bytes from the pipe.

When a task calls **_io_read()**, the function returns when the driver has read the specified number of bytes from the pipe. If the pipe does not contain enough bytes, the task blocks.

Because of this blocking behavior, an application cannot call **_io_read()** and **_io_write()** from an interrupt service routine.

4.7 I/O Control Commands

This section describes the I/O control commands that you use when you call **_io_ioctl()**. They are defined in *io_pipe.h*.

Command	Description
PIPE_IOCTL_GET_SIZE	Get the size of the pipe in chars.
PIPE_IOCTL_FULL	Determine whether the pipe is full (TRUE indicates full).
PIPE_IOCTL_EMPTY	Determine whether the pipe is empty (TRUE indicates empty).
PIPE_IOCTL_RE_INIT	Delete all the data from the pipe.
PIPE_IOCTL_CHAR_AVAIL	Determine whether the data is available (TRUE indicates that the data is available).
PIPE_IOCTL_NUM_CHARS_FULL	Get the number of *char*s in the pipe.
PIPE_IOCTL_NUM_CHARS_FREE	Get the amount of free chars in the pipe.

Chapter 5 Serial-Device Families

5.1 Overview

This section describes the information that applies to all serial-device drivers that accompany MQX. The subfamilies of the drivers include:

- Serial interrupt-driven I/O
- Serial-polled I/O

5.2 Source Code Location

Driver	Location
Serial interrupt-driven	`source\io\serial\int`
Serial polled	`source\io\serial\polled`

5.3 Header Files

To use a serial-device driver, include the header file from *source\io\serial* in your application or in the BSP file *bsp.h*. Use the header file according to the following table.

Driver	Header File
Serial interrupt-driven	`serial.h`
Serial polled	`serial.h`

The files *serinprv.h* and *serplprv.h* contain private constants and data structures that serial-device drivers use. You must include this file if you recompile a serial-device driver. You may also want to look at the file as you debug your application.

5.4 Installing Drivers

Each serial-device driver provides an installation function that either the BSP or the application calls. The function then calls **_io_dev_install()** internally. Different installation functions exist for different UART hardware modules. Please see the BSP initialization code in *init_bsp.c* for functions suitable for your hardware (*xxxx* in the function names below).

Driver	Installation Function
Interrupt-driven	_xxxx_serial_int_install()
Polled	_xxxx_serial_polled_install()

5.4.1 Initialization Records

Each installation function requires a pointer to the initialization record to be passed to it. This record is used to initialize the device and software when the device is first opened. The record is unique to each possible device, and the fields required along with initialization values are defined in the device-specific header files.

Synopsis for kinetis, mcf51jf and mcf51qm family

```
#include <serl_kuart.h>
typedef struct kuart_init_struct
{
   uint_32                          QUEUE_SIZE;
   uint_32                          DEVICE;
   uint_32                          CLOCK_SPEED;
   uint_32                          BAUD_RATE;
   uint_32                          RX_TX_VECTOR;
   uint_32                          ERR_VECTOR;
   uint_32                          RX_TX_PRIORITY;
   uint_32                          ERR_PRIORITY;
#if MQX_ENABLE_LOW_POWER
   CM_CLOCK_SOURCE                  CLOCK_SOURCE;
   KUART_OPERATION_MODE_STRUCT_CPTR OPERATION_MODE;
#endif
} KUART_INIT_STRUCT, _PTR_ KUART_INIT_STRUCT_PTR;
```

Parameters

QUEUE_SIZE - The size of the queues to buffer incoming/outgoing data.

DEVICE - The device to initialize.

CLOCK_SPEED - The clock speed of cpu.

BAUD_RATE - The baud rate for the channel.

RX_TX_VECTOR - RX / TX interrupt vector.

ERR_VECTOR - ERR interrupt vector.

RX_TX_PRIORITY - RX / TX interrupt vector priority.

ERR_PRIORITY - ERR interrupt vector priority.

CLOCK_SOURCE - Clock source when low power is enabled.

OPERATION_MODE - Low power operation mode when low power is enabled.

Synopsis for mcf51XX family (except mcf51jf and mcf51qm)

```
#include <serl_mcf51xx.h>
typedef struct mcf51xx_sci_init_struct
{
   uint_32 QUEUE_SIZE;
   uint_32 DEVICE;
   uint_32 CLOCK_SPEED;
   uint_8  SCIC1_VALUE;
   uint_8  SCIC2_VALUE;
```

```
    uint_8  SCIC3_VALUE;
    uint_32 BAUD_RATE;
    uint_32 RX_VECTOR;
    uint_32 TX_VECTOR;
    uint_32 ER_VECTOR;
} MCF51XX_SCI_INIT_STRUCT, _PTR_ MCF51XX_SCI_INIT_STRUCT_PTR;
```

Parameters

QUEUE_SIZE - The size of the queues to buffer incoming/outgoing data.

DEVICE - The device to initialize.

CLOCK_SPEED - The clock speed of cpu.

SCIC1_VALUE - The value for the SCIxC1 (SCI Control Register 1).

SCIC2_VALUE - The value for the SCIxC2 (SCI Control Register 2).

SCIC3_VALUE - The value for the SCIxC3 (SCI Control Register 3).

BAUD_RATE - The baud rate for the channel.

RX_VECTOR - RX interrupt vector.

TX_VECTOR - TX interrupt vector.

ER_VECTOR - ERROR interrupt vector.

Synopsis for mcf52XX, mcf53XX, mcf54XX family (example for mcf52XX)

```
#include <serl_mcf52xx.h>
typedef struct mcf52XX_uart_serial_init_struct
{
    uint_32       QUEUE_SIZE;
    uint_32       DEVICE;
    uint_32       CLOCK_SPEED;
    uint_32       VECTOR;
    _int_level    LEVEL;
    _int_priority SUBLEVEL;
    uint_32       UMR1_VALUE;
    uint_32       UMR2_VALUE;
    uint_32       BAUD_RATE;
} MCF52XX_UART_SERIAL_INIT_STRUCT, _PTR_ MCF52XX_UART_SERIAL_INIT_STRUCT_PTR;
```

Parameters

QUEUE_SIZE - The size of the queues to buffer incoming/outgoing data.

DEVICE - The device to initialize.

CLOCK_SPEED - The clock speed of cpu.

VECTOR - The interrupt vector to use if interrupt driven.

LEVEL - The interrupt level to use if interrupt driven.

SUBLEVEL - The sub-level within the interrupt level to use if interrupt driven.

UMR1_VALUE - The value for the UMR 1 (Uart Mode Register 1).

UMR2_VALUE - The value for the UMR 2 (Uart Mode Register 2).

BAUD_RATE - The baud rate for the channel.

Synopsis for pxs20, pxs30

```
#include <serl_linflexd.h>
typedef struct linflexd_serial_init_struct
{
    uint_32     DEVICE;
    _mqx_uint   QUEUE_SIZE;
    uint_32     CLOCK_SPEED;
    uint_32     BAUD_RATE;
    _mqx_uint   BITS_PER_CHARACTER;
    _mqx_uint   PARITY;
    _mqx_uint   STOP_BITS;
    uint_32     PRIORITY;
} LINFLEXD_SERIAL_INIT_STRUCT, _PTR_ LINFLEXD_SERIAL_INIT_STRUCT_PTR;
```

Parameters

DEVICE - The device to initialize.

QUEUE_SIZE - The size of the queues to buffer incoming/outgoing data.

CLOCK_SPEED - The clock speed of cpu.

BAUD_RATE - The baud rate for the channel

BITS_PER_CHARACTER - The number of bits in a character.

PARITY - The parity to initialize the channel to.

STOP_BITS - The number of stop bits.

PRIORITY - Interrupt priority.

Synopsis for mpc5125

```
#include <serl_mpc5125.h>
typedef struct mpc5125_serial_init_struct
{
   uint_32   DEVICE;
   _mqx_uint QUEUE_SIZE;
   _mqx_uint RX_QUEUE_ALARM;
   _mqx_uint TX_QUEUE_ALARM;
   uint_32   BAUD_RATE;
   _mqx_uint BITS_PER_CHARACTER;
   _mqx_uint PARITY;
   _mqx_uint STOP_BITS;
   uint_32   (_CODE_PTR_ CLOCK_SPEED)(void);
} MPC5125_SERIAL_INIT_STRUCT, _PTR_ MPC5125_SERIAL_INIT_STRUCT_PTR;
```

Parameters

DEVICE - The com port offset from MBAR.

QUEUE_SIZE - The MQX serial I/O queue size to use to buffer Rx/Tx data.

RX_QUEUE_ALARM - Uart device driver Rx alarm settings for RxRDY.

TX_QUEUE_ALARM - Uart device driver Tx alarm settings for TxRDY.
BAUD_RATE - The baud rate for the channel.
BITS_PER_CHARACTER - The number of bits in a character.
PARITY - The parity to initialize the channel to.
STOP_BITS - The number of stop bits.
(_CODE_PTR_ CLOCK_SPEED)(void) - Function which returns clock speed to the UART
VECTOR - The com ports vector number (interrupt table index).

Example

The following is an example for the MCF52xx family of microcontrollers as it can be found in the appropriate BSP code (see for example the *init_uart0.c* file).

```
const MCF52XX_UART_SERIAL_INIT_STRUCT _bsp_uart0_init = {
   /* queue size        */ BSPCFG_UART0_QUEUE_SIZE,
   /* Channel           */ MCF52XX_IO_UART0,
   /* Clock Speed       */ BSP_SYSTEM_CLOCK,
   /* Interrupt Vector  */ BSP_UART0_INT_VECTOR,
   /* Interrupt Level   */ BSP_UART0_INT_LEVEL,
   /* Interrupt Sublevel */ BSP_UART0_INT_SUBLEVEL,
   /* UMR1 Value        */ MCF52XX_UART_UMR1_NO_PARITY |
                           MCF52XX_UART_UMR1_8_BITS,
   /* UMR2 Value        */ MCF52XX_UART_UMR2_1_STOP_BIT,
   /* Baud rate         */ BSPCFG_UART0_BAUD_RATE
};
```

5.5 Driver Services

The serial device driver provides these services:

API	Calls	
	Interrupt-driven	Polled
_io_fopen()	_io_serial_int_open()	_io_serial_polled_open()
_io_fclose()	_io_serial_int_close()	_io_serial_polled_close()
_io_read()	_io_serial_int_read()	_io_serial_polled_read()
_io_write()	_io_serial_int_write()	_io_serial_polled_write()
_io_ioctl()	_io_serial_int_ioctl()	_io_serial_polled_ioctl()

5.6 I/O Open Flags

This section describes the flag values you can pass when you call **_io_fopen()** for a particular interrupt-driven or polled serial-device driver. They are defined in *serial.h*.

Command	Description
IO_SERIAL_RAW_IO	No processing of I/O done.
IO_SERIAL_XON_XOFF	Software flow control enabled.
IO_SERIAL_TRANSLATION	Translation of: outgoing \n to CRLF incoming CR to \n incoming backspace outputs backspace space backspace and drops the input.
IO_SERIAL_ECHO	Echoes incoming characters.
IO_SERIAL_HW_FLOW_CONTROL	Enables hardware flow control (RTS/CTS) where available.
IO_SERIAL_NON_BLOCKING	Opens the serial driver in non blocking mode. In this mode the _io_read() function doesn't wait till the receive buffer is full. It immediately returns received characters and number of received characters.
IO_SERIAL_HW_485_FLOW_CONTROL	Enables hardware support for RS485 if it is available on target processor. Target HW automatically asserts RTS signal before transmitting the message and deasserts it after transmission is done.

5.7 I/O Control Commands

This section describes the I/O control commands that you use when you call **_io_ioctl()** for a particular interrupt-driven or polled serial-device driver. Each of these commands may or may not be implemented by a specific device driver. They are defined in *serial.h*.

Command	Description
IO_IOCTL_SERIAL_CLEAR_STATS	Clear the statistics.
IO_IOCTL_SERIAL_GET_BAUD	Get the BAUD rate.
IO_IOCTL_SERIAL_GET_CONFIG	Get the device configuration.
IO_IOCTL_SERIAL_GET_FLAGS	Get the flags.
IO_IOCTL_SERIAL_GET_STATS	Get the statistics.
IO_IOCTL_SERIAL_SET_BAUD	Set the BAUD rate.
IO_IOCTL_SERIAL_SET_FLAGS	Set the flags.
IO_IOCTL_SERIAL_TRANSMIT_DONE	Returns TRUE if output ring buffer empties.

Command	Description
IO_IOCTL_SERIAL_GET_HW_SIGNAL	Returns hardware signal value.
IO_IOCTL_SERIAL_SET_HW_SIGNAL	Asserts the hardware signals specified.
IO_IOCTL_SERIAL_CLEAR_HW_SIGNAL	Clears the hardware signals specified.
IO_IOCTL_SERIAL_SET_DATA_BITS	Sets the number of data bits in the characters.
IO_IOCTL_SERIAL_GET_DATA_BITS	Gets the number of data bits in the characters.
IO_IOCTL_SERIAL_SET_STOP_BITS	Sets the number of stop bits in the character.
IO_IOCTL_SERIAL_GET_STOP_BITS	Gets the number of stop bits in the character.
IO_IOCTL_SERIAL_TX_DRAINED	Returns TRUE if there are no transmit characters in the FIFOs or in the software rings.
IO_IOCTL_SERIAL_DISABLE_RX	Disables or enables UART receiver.
IO_IOCTL_SERIAL_WAIT_FOR_TC	Waits until the transmission complete (TC) flag is set. This IO control command uses busy-wait loop and does not check the state of internal serial driver buffers. In case the application is waiting for whole buffer transmission use together with fflush() command, see example below.
IO_IOCTL_SERIAL_CAN_TRANSMIT	Returns 1 in ioctl parameter when the there's a room in HW transmit buffer for another character, returns 0 otherwise.
IO_IOCTL_SERIAL_CAN_RECEIVE	Returns 1 in ioctl parameter when there's at least one character in input HW buffer, returns 0 otherwise.
IO_IOCTL_SERIAL_GET_PARITY	Returns in ioctl parameter the type of parity that is currently configured.
IO_IOCTL_SERIAL_SET_PARITY	Sets the given type of parity.

5.8 I/O Hardware Signals

This section describes the hardware signal values you can pass when you call **_io_ioctl()** with the HW_SIGNAL commands. The signals may or may not be present depending upon the hardware implementation. They are defined in *serial.h*.

Signal	Description
IO_SERIAL_CTS	Hardware CTS signal
IO_SERIAL_RTS	Hardware RTS signal
IO_SERIAL_DTR	Hardware DTR signal

IO_SERIAL_DSR	Hardware DSRsignal
IO_SERIAL_DCD	Hardware DCD signal
IO_SERIAL_RI	Hardware RI signal

5.9 I/O Stop Bits

This section describes the stop-bit values you can pass when you call **_io_ioctl()** with the IOCTL STOP BITS commands. They are defined in *serial.h.*

Signal	Description
IO_SERIAL_STOP_BITS_1	1 stop bit
IO_SERIAL_STOP_BITS_1_5	1 1/2 stop bits
IO_SERIAL_STOP_BITS_2	2 stop bits

5.10 I/O Parity

This section describes the parity values you can pass when you call **_io_ioctl()** with the IOCTL PARITY commands. They are defined in *serial.h.*

Signal	Description
IO_SERIAL_PARITY_NONE	No parity
IO_SERIAL_PARITY_ODD	Odd parity
IO_SERIAL_PARITY_EVEN	Even parity
IO_SERIAL_PARITY_FORCE	Force parity
IO_SERIAL_PARITY_MARK	Set parity bit to mark
IO_SERIAL_PARITY_SPACE	Set parity bit to space

5.11 RS485 Support in Serial Device

If the RS485 communication is required, the following steps have to be done:

1. Open the serial device. If the MCU supports hardware flow control, use IO_SERIAL_HW_485_FLOW_CONTROL flag.
2. Disable transmitter if needed. This can be required if hardware echo is hardwired.
3. If the IO_SERIAL_HW_485_FLOW_CONTROL is not supported, select an appropriate GPIO pin and enable RS485 driver transmitter.
4. Send a message.
5. Wait for an empty sending queue. Use fflush().
6. Wait for the transfer complete flag. Use IO_IOCTL_SERIAL_WAIT_FOR_TC.

7. For devices without IO_SERIAL_HW_485_FLOW_CONTROL, de-assert the GPIO pin.
8. Enable receiver if it was disabled before.

Example

The following example shows how to initialize and control the RS485 communication.

```
MQX_FILE_PTR rs485_dev = NULL;
char  data_buffer[] = "RS485 send example";
boolean disable_rx = TRUE;

/*
** If mcu has hardware support for RTS pin drive (e.g. k60n512),
** open line with IO_SERIAL_HW_485_FLOW_CONTROL flag
*/
#if (HAS_485_HW_FLOW_CONTROL)
/* HW 485 flow control on chip*/
rs485_dev = fopen( RS485_CHANNEL, ( char const * )   IO_SERIAL_HW_485_FLOW_CONTROL );
#else
/* HW 485 flow not available on chip */
rs485_dev  = fopen( RS485_CHANNEL, NULL );
#endif

/*
** Half duplex, two wire mode. Use only if disable receiver in
** transmit is desired
*/
ioctl( rs485_dev, IO_IOCTL_SERIAL_DISABLE_RX, &disable_rx );

#if !(HAS_485_HW_FLOW_CONTROL)
/*
** User written function for flow control by GPIO pin - handle RTS
** or other signal to drive RS485 HW driver
*/
#endif

/* write data */
write( rs485_dev, data_buffer, strlen(data_buffer) );
```

```
/* empty queue - not needed for polled mode */
fflush( rs485_dev );

/* wait for transfer complete flag */
ioctl( rs485_dev, IO_IOCTL_SERIAL_WAIT_FOR_TC, NULL );

/* half duplex, two wire */
/* if receiver was disabled before, enable receiver again */
disable_rx = FALSE;
ioctl( rs485_dev, IO_IOCTL_SERIAL_DISABLE_RX, &disable_rx ) ;

#if !( HAS_485_HW_FLOW_CONTROL )
/*
** User written function for flow control by GPIO pin - handle RTS
*/
#endif
```

5.12 Error Codes

No additional error codes are generated.

5.13 Low Power Support

5.13.1 Overview

The MQX Low Power support for serial device driver is designed to reduce system power consumption by disabling assigned pins, the peripheral clock, or the peripheral itself according to a setting predefined for several operation modes. Another goal is the application-transparent adaptation to a system frequency change. The mode and frequency change is handled by the MQX Low Power Manger (LPM) component. This sections describes a setting specific to Serial Driver.

The low power functionality is currently implemented for Kinetis and ColdFire+ based BSPs only. You have to define MQX_ENABLE_LOW_POWER as nonzero in the *user_config.h* file and compile BSP with this setting before the usage.

Both polled and interrupt serial drivers are registered at LPM during their installation. Similarly, the uninstall function unregisters them from the LPM. The registration contains callbacks that are used by the LPM to notify the serial driver about operation mode and clock configuration change. The serial peripheral device behavior is changed within these callbacks to reflect the settings defined in the `_bsp_sciX_operation_modes` configuration structures. The configuration structures are defined for each serial peripheral channel (`sciX`) in the *init_spi.c* file. In addition to the power consumption settings, these structures also allow configuring the wakeup behavior of the serial peripheral device in the interrupt mode.

5.13.2 Data Type Definitions

The data types related to the low power functionality of the serial driver are defined in a corresponding platform specific serial header file. Following definitions apply to Kinetis platform and can be found in *serl_kuart.h* file in the serial driver directory.

The serial operation mode structure allows enabling or disabling the serial peripheral device functionality and parametrizing its wakeup behavior in a particular operation mode.

```
typedef struct kuart_operation_mode_struct
{
    uint_8      FLAGS;
    uint_8      WAKEUP_BITS;
    uint_8      MA1;
    uint_8      MA2;
} KUART_OPERATION_MODE_STRUCT, _PTR_ KUART_OPERATION_MODE_STRUCT_PTR;
typedef const KUART_OPERATION_MODE_STRUCT _PTR_
KUART_OPERATION_MODE_STRUCT_CPTR;
```

The following flags can be used to specify functionality of a serial peripheral device within a serial operation mode structure.

FLAGS	Description
IO_PERIPHERAL_PIN_MUX_ENABLE IO_PERIPHERAL_PIN_MUX_DISABLE	Enables a set peripheral pins multiplexer to serial driver functionality, or disables pins associated with a particular serial peripheral device. If neither of these flags is specified, the pin multiplexer setting is not changed. If both flags are specified, the disable flag has precedence. Implementation is in the BSP specific *_bsp_serial_io_init()* function, in *init_gpio.c*.
IO_PERIPHERAL_CLOCK_ENABLE IO_PERIPHERAL_CLOCK_DISABLE	Enables/disables input clock of the serial peripheral device. If neither of these flags is specified, the pin clock setting is not changed. If both flags are specified, the disable flag has precedence. Implementation is in the BSP specific *_bsp_serial_io_init()* function, in *init_gpio.c*.

IO_PERIPHERAL_MODULE_ENABLE	If his flag is used, the serial module is enabled and internal registers are available. If it is not specified, the module is disabled.
IO_PERIPHERAL_WAKEUP_ENABLE	If specified, enables serial peripheral module to wakeup CPU core according to WAKEUP_BITS setting which are given in the serial operation mode behavior structure. If not specified, the serial driver wakeup functionality is disabled and ISR are not serviced in the CPU core sleep modes. This functionality is available in the interrupt serial driver mode only.
IO_PERIPHERAL_WAKEUP_SLEEPONEXIT_DISABLE	This flag specifies behavior of the CPU core after the return from wakeup ISR. If the flag is used, the CPU core exits the sleep mode. If the flag is not used, the CPU core stays in the sleep mode and waits for the next wakeup event. In both cases, the ISR is executed. Setting of this flag is relevant only if IO_PERIPHERAL_WAKEUP_ENABLE is used.

Combination of the following register bits can be used to set up wakeup functionality of the serial peripheral device on the Kinetis platform. See the processor Reference Manual for details.

WAKEUP_BITS	Description
UART_C2_RWU_MASK	Places receiver instantly into standby state.
UART_C1_WAKE_MASK	Wakeup method select. When specified, address mark match instead of idle line wakeup is used.
UART_C1_ILT_MASK	Idle line type select. When used, idle character starts after stop bit instead of start bit.
UART_C4_MAEN1_MASK	Forces serial device to compare incoming bytes with value of its MA1 register and to wakeup on match.
UART_C4_MAEN2_MASK	Forces serial device to compare incoming bytes with value of its MA2 register and to wakeup on match.

MAx	Description
MA1	Wakeup match addresses when wakeup UART_C4_MAEN1_MASK method is selected. See Kinetis Reference maual for details.
MA2	Wakeup match addresses when wakeup UART_C4_MAEN2_MASK method is selected. See Kinetis Reference Manual for details.

If the low power feature is enabled, the serial initialization structure is extended by a clock source ID and a pointer to the behavior definitions in all operation modes.

```
typedef struct kuart_init_struct
{
   uint_32 QUEUE_SIZE;
   uint_32 DEVICE;
   uint_32 CLOCK_SPEED;
   uint_32 BAUD_RATE;
   uint_32 RX_TX_VECTOR;
   uint_32 ERR_VECTOR;
   uint_32 RX_TX_PRIORITY;
   uint_32 ERR_PRIORITY;

#if MQX_ENABLE_LOW_POWER
   CM_CLOCK_SOURCE                          CLOCK_SOURCE;
   KUART_OPERATION_MODE_STRUCT_CPTR         OPERATION_MODE;
#endif

} KUART_INIT_STRUCT, _PTR_ KUART_INIT_STRUCT_PTR;
typedef const KUART_INIT_STRUCT _PTR_ KUART_INIT_STRUCT_CPTR;
```

5.13.3 Default BSP Settings

There are two macros in the main BSP header file that provide the dependency level for polled and interrupt serial drivers. The order of notifications of all drivers registered at LPM is based on these levels. For pre-notifications, the lower dependency level drivers are processed first and the order of registration is used for the same dependency levels. For post-notifications or for rollback, in case of failure, the order is reversed.

```
#define BSP_LPM_DEPENDENCY_LEVEL_SERIAL_POLLED (30)
#define BSP_LPM_DEPENDENCY_LEVEL_SERIAL_INT    (31)
```

The serial initialization file in the BSP is extended by the operation mode behavior structures for each peripheral device. The following demonstrates the behavior definitions for SCI3 (ttyd). The peripheral itself, associated pins, and the peripheral clock are enabled in operation modes RUN, WAIT, and SLEEP. In the SLEEP operation mode, the wakeup on idle-line counting from start bit is enabled. Also, CPU core wakes up after the serial ISR. The SCI3 peripheral, its input clock, and associated pins are completely disabled in the STOP operation mode. The described setting is achieved by the configuration bellow. It can be changed at any time to match your application requirements.

```
const KUART_OPERATION_MODE_STRUCT
_bsp_sci3_operation_modes[LPM_OPERATION_MODES] =
{
    /* LPM_OPERATION_MODE_RUN */
    {
        IO_PERIPHERAL_PIN_MUX_ENABLE | IO_PERIPHERAL_CLOCK_ENABLE |
        IO_PERIPHERAL_MODULE_ENABLE,
        0,
        0,
        0
    },

    /* LPM_OPERATION_MODE_WAIT */
    {
        IO_PERIPHERAL_PIN_MUX_ENABLE | IO_PERIPHERAL_CLOCK_ENABLE |
        IO_PERIPHERAL_MODULE_ENABLE,
        0,
        0,
        0
    },

    /* LPM_OPERATION_MODE_SLEEP */
    {
        IO_PERIPHERAL_PIN_MUX_ENABLE | IO_PERIPHERAL_CLOCK_ENABLE |
        IO_PERIPHERAL_MODULE_ENABLE | IO_PERIPHERAL_WAKEUP_ENABLE |
        IO_PERIPHERAL_WAKEUP_SLEEPONEXIT_DISABLE,
        0,
        0,
        0
    },

    /* LPM_OPERATION_MODE_STOP */
    {
        IO_PERIPHERAL_PIN_MUX_DISABLE | IO_PERIPHERAL_CLOCK_DISABLE,
        0,
        0,
        0
    }
};

const KUART_INIT_STRUCT _bsp_sci3_init = {
   /* queue size         */ BSPCFG_SCI3_QUEUE_SIZE,
```

```
    /* Channel            */ 3,
    /* Clock Speed        */ BSP_BUS_CLOCK,
    /* Baud rate          */ BSPCFG_SCI3_BAUD_RATE,
    /* RX/TX Int vect     */ INT_UART3_RX_TX,
    /* ERR Int vect       */ INT_UART3_ERR,
    /* RX/TX priority     */ 3,
    /* ERR priority       */ 4
#if MQX_ENABLE_LOW_POWER
    ,
    /* Clock source       */ CM_CLOCK_SOURCE_BUS,
    /* LPM operation info */ _bsp_sci3_operation_modes
#endif
};
```

5.13.4 Remarks

Even if the behavior structure (KUART_OPERATION_MODE_STRUCT) states that the serial driver peripheral HW module needs to be enabled, the peripheral is not touched until the driver is opened. It is an application responsibility to flush and stop using the serial driver before switching to the low power mode where the serial peripheral module or its clock is disabled. Otherwise, unexpected results may occur.

Also, enabling both polled and interrupt drivers over the same peripheral device can lead to unexpected results, because both drivers are registered at LPM and the low power change happens twice over the same HW module. This can lead to disable/enable conflict when only one of the drivers is opened and dependency levels of their registrations at LPM create a wrong order of notifications for polled and interrupt serial driver.

Chapter 6 Simple Memory Driver

6.1 Overview

The simple memory driver provides an I/O device that writes to a configured block of memory. All normal operations such as read, write, and seek work properly. The read and write operations are locked with a semaphore so that the entire operation can complete uninterruptedly.

6.2 Source Code Location

The source code for the simple memory driver is in `source\io\io_mem`.

6.3 Header Files

For the simple memory driver, include the header file *io_mem.h* in your application or in the BSP file *bsp.h*.

The file *iomemprv.h* contains private constants and data structures that the driver uses. You must include this file if you recompile the driver. You may also want to look at the file as you debug your application.

6.4 Driver Services

The simple memory driver provides these services:

API	Calls
_io_fopen()	_io_mem_open()
_io_fclose()	_io_mem_close()
_io_read()	_io_mem_read()
_io_write()	_io_mem_write()
_io_ioctl()	_io_mem_ioctl()

6.5 Installing Drivers

The simple memory driver provides an installation function that either the BSP or the application calls. The function installs the **_io_mem** family of functions and calls **_io_dev_install()**.

```
_mqx_uint _io_mem_install
   (
      /* [IN] A string that identifies the device for fopen */
      char_ptr            identifier,
      /* [IN] the starting address of the device in memory */
      pointer             base_address,
      /* [IN] the total size of the device */
      _file_size          size
   )
```

6.6 I/O Control Commands

This section describes the I/O control commands you need to use when you call **_io_ioctl().** They are defined in *io_mem.h.*

Command	Description
IO_MEM_IOCTL_GET_BASE_ADDRESS	The base address of the memory block written to by this device.
IO_MEM_IOCTL_GET_TOTAL_SIZE	The total size of the memory block written to by this device.
IO_MEM_IOCTL_GET_DEVICE_ERROR	The error code stored in the file descriptor.

6.7 Error Codes

No additional error codes are provided by this driver.

Chapter 7 GPIO Driver

7.1 Overview

GPIO driver creates the hardware abstraction layer so that the application can use input or output pins.

The GPIO API is divided into two parts:

- Hardware-independent generic driver
- Hardware-dependent layer called hardware-specific driver

7.2 Source Code Location

Driver	Location
GPIO generic driver	`source\io\gpio`
GPIO hardware-specific driver	`source\io\gpio\<CPU_name>`

7.3 Header Files

To use GPIO driver, include the header files from the *lib* directory in your application.

Driver	Header file
GPIO generic driver	`io_gpio.h`
GPIO hardware-specific driver	`io_gpio_<CPU_name>.h`

7.4 Installing Drivers

Each GPIO driver provides an installation function that either the BSP or the application calls. The function then calls **_io_dev_install()** internally. Usually, **_io_gpio_install()** installation function is called from *init_bsp.c* if enabled by **BSPCFG_ENABLE_GPIO** configuration option in *user_config.h*.

```
_io_gpio_install("gpio:");
```

7.5 Opening GPIO Device

To access GPIO pins, it is necessary to open the GPIO device with a parameter specifying the set of pins to be used. The direction (input or output) of the whole pin set must be defined as shown in the following example:

```
file = fopen("gpio:input", &pin_table);
```

The *pin_table* is an array of *the GPIO_PIN_STRUCT* ending with *GPIO_LIST_END*. To describe a pin, the header file definitions must be used. The following expression is used to describe a pin:

```
<port_name> | <pin_number> | <additional_flags>
```

where:

Parameter	Description
<port_name>	Port name specified in the GPIO hardware-specific header file.
<pin_number>	Pin number specified in the GPIO generic header file.
<additional_flags>	Flags for pin behavior. General (see GPIO generic header file) or hardware-specific (see GPIO hardware-specific header file): • GPIO_PIN_STATUS_0 ... for the gpio:output device; this flag clears the pin state after opening device file. • GPIO_PIN_STATUS_1... for the gpio:output device; this flag sets the pin state after opening device file. • GPIO_PIN_IRQ_RISING... for the gpio:input device; this flag enables the pin status change interrupt callback function which is set by GPIO_IOCTL_SET_IRQ_FUNCTION command and allows the interrupt callback function being called when the rising edge occurs. • GPIO_PIN_IRQ_FALLING... for the gpio:input device; this flag enables the pin status change interrupt callback function which is set by GPIO_IOCTL_SET_IRQ_FUNCTION command and allows the interrupt callback function being called when the falling edge occurs. • GPIO_PIN_IRQ ... this is an obsolete flag identical to the GPIO_PIN_IRQ_RISING flag.

An example of the *pin_table* initialization structure:

```
const GPIO_PIN_STRUCT pin_table[] = {
        GPIO_PORT_NQ | GPIO_PIN5 | GPIO_PIN_IRQ,
        GPIO_PORT_TC | GPIO_PIN3,
        GPIO_LIST_END
   };
```

NOTE

The pin can be used only by one file. Otherwise, a NULL pointer is returned by **fopen**.

7.6 Driver Services

The GPIO device driver provides the following services:

API	Calls
_io_fopen()	_gpio_open()
_io_fclose()	_gpio_close()
_io_ioctl()	_gpio_ioctl()

7.7 Generic IOCTL Commands

This section describes the I/O control commands that you use when you call **_io_ioctl()**.

Command	Description
GPIO_IOCTL_ADD_PINS	Adds pins to the file. The parameter is GPIO_PIN_STRUCT array.
GPIO_IOCTL_WRITE_LOG1	Sets output pins. If the parameter is GPIO_PIN_STRUCT array, the driver sets all pins specified. Pin list passed in the array must be a subset of file pins. If the parameter is NULL, all file pins will be set.
GPIO_IOCTL_WRITE_LOG0	Clears output pins. If the parameter is GPIO_PIN_STRUCT array, driver clears all pins specified. Pin list passed in the array must be a subset of file pins. If the parameter is NULL, all file pins will be cleared.
GPIO_IOCTL_WRITE	Sets or clears output pins according to GPIO_PIN_STRUCT array. Pin list passed in the array must be a subset of file pins. An Array contains status of each pin by using GPIO_PIN_STATUS_0 and GPIO_PIN_STATUS_1 flags.
GPIO_IOCTL_READ	Reads status of input pins and updates the GPIO_PIN_STRUCT array. Pin list passed in the array must be a subset of file pins. Uses the GPIO_PIN_STATUS mask on each item of the returned GPIO_PIN_STRUCT array to get the state of the pin.
GPIO_IOCTL_SET_IRQ_FUNCTION	Sets the callback function which is invoked for any IRQ event coming from any file pin.
GPIO_IOCTL_ENABLE_IRQ	Enables IRQ functionality for all IRQ pins in the file.
GPIO_IOCTL_DISABLE_IRQ	Disables IRQ functionality for all IRQ pins in the file.

The following is an example of using IOCTL command for the GPIO driver:

Set all pins attached to the file:

```
ioctl(file, GPIO_IOCTL_WRITE_LOG1, NULL);
```

Read pin status of input file to *read_pin_table*:

```
if(ioctl(file, GPIO_IOCTL_READ, &read_pin_table) == IO_OK)
{
        if((read_pin_table[0] & GPIO_PIN_STATUS) == GPIO_PIN_STATUS_1)
        {
                // first pin in the table is set
        }
}
```

7.8 Hardware-Specific IOCTL Commands

Hardware-specific commands are used to handle specific MCU behavior and hardware performance. These commands are not portable to other processor.

No hardware-specific commands are implemented yet.

7.9 Error Codes

No additional error codes are generated.

Chapter 8 ADC Driver

8.1 Overview

This section describes the ADC device drivers that accompany the Freescale MQX.

8.2 Source Code Location

Driver	Location
ADC generic driver	`source\io\adc`
ADC hardware-specific driver	`source\io\adc\<CPU_name>`

8.3 Header Files

To use the ADC device driver, include the header file from *source\io\adc* in your application or in the BSP file *bsp.h*. Use the header file according to the following table:

Driver	Header File
ADC driver	`adc.h`

The file *adc_prv.h* contains private constants and data structures that ADC device driver uses.

8.4 Installing ADC Driver

ADC device driver provides an installation function **_io_adc_install()** that either the BSP or the application calls. The function then calls **_io_dev_install()** internally. Usually **_io_adc_install()** installation function is called from *init_bsp.c* if enabled by **BSPCFG_ENABLE_ADC** configuration option in *user_config.h*.

Example of the **_io_adc_install** function call is as follows:

```
_io_adc_install("adc1:", (pointer) adc_init_struct);
```

The `adc_init_struct` is a pointer to an initialization structure containing information for ADC driver. For HW specific drivers which do not support initialization structures, the NULL pointer is passed instead.

8.4.1 Initialization Records

Each installation function requires a pointer to initialization record to be passed to it. This record is used to initialize the device and software when the device is opened for the first time. The record is unique to every device and the fields required along with initialization values are defined in the device-specific header files.

Synopsis for KADC (Kinetis family, mcf51jm and mcf51qm)

```
#include <adc_kadc.h>
typedef struct kadc_install_struct
{
uint_8           ADC_NUMBER;
ADC_CLOCK_SOURCE CLOCK_SOURCE;
ADC_CLOCK_DIV    CLOCK_DIV;
ADC_HSC          SPEED;
ADC_LPC          POWER;
uint_8_ptr       CALIBRATION_DATA_PTR;
uint_32          ADC_VECTOR;
uint_32          ADC_PRIORITY;
KPDB_INIT_STRUCT const *PDB_INIT;
} KADC_INIT_STRUCT, _PTR_ KADC_INIT_STRUCT_PTR
```

Parameters

ADC_NUMBER - Number of ADC peripheral. Use adc_t enum from PSP.

CLOCK_SOURCE - Clock source. Use enum defined in the KADC header.

CLOCK_DIVISOR - Clock divisor. Use enum defined in the KADC header.

SPEED - High speed control. See ADC_HSC enum.

POWER - Low power control. See ADC_LPC enum.

CALIBRATION_DATA_PTR - Pointer to calibration data. Contains initialization values for calibration related registers.

ADC_VECTOR - ADC interrupt vector.

ADC_PRIORITY - Priority of the ADC interrupt.

PDB_INIT - Pointer to KPDB init structure to initialize programmable delay block.

Synopsis for mcf51ag, mcf51em, mcf51je and mcf51mm

Example of structure for mcf51ag:

```
#include <adc_mcf51ag.h>
typedef struct adc_install_struct
{
uint_8           ADC_NUMBER;
ADC_CLOCK_SOURCE CLOCK_SOURCE;
ADC_CLOCK_DIV    CLOCK_DIV;
uint_8_ptr       CALIBRATION_DATA_PTR;
uint_32          ADC_VECTOR;
} MCF51AG_ADC_INIT_STRUCT, _PTR_ MCF51AG_ADC_INIT_STRUCT_PTR;
```

Parameters

ADC_NUMBER - Number of ADC peripheral. Use adc_t enum from PSP.

CLOCK_SOURCE - Clock source. Use enum defined in the header of the driver for given platform.

CLOCK_DIVISOR - Clock divisor. Use enum defined in the header of the driver for given platform.

CALIBRATION_DATA_PTR - Pointer to calibration data. Contains initialization values for calibration related registers.

ADC_VECTOR - ADC interrupt vector.

Synopsis for mcf522xx and mcf544xx

There is no ADC init structure for these plaftorms. A NULL pointer should be passed to *_io_adc_install* function.

8.5 Driver Services

The ADC device driver provides these services:

API	Calls
_io_fopen()	_adc_open()
_io_fclose()	_adc_close()
_io_read()	_adc_read()
_io_write()	_adc_write()
_io_ioctl()	_adc_ioctl()

8.5.1 Opening ADC Device

The device open function requires a pointer to initialization record. This record is used to initialize the ADC module and software driver when the device is first opened.

The following is an example for the MCF52xx family of microcontrollers as it can be found in the appropriate example code (see the */mqx/example/adc/adc.c* file).

```
/* ADC device init struct */
const ADC_INIT_STRUCT adc_init = {
    ADC_RESOLUTION_DEFAULT,     /* resolution */
};


f = fopen("adc:", (const char*)&adc_init);
```

The table below describes flags you can pass when you call **fopen()** for ADC device. They are defined in *adc_<CPU_name>.h.*

Flag Value	Description
ADC_RESOLUTION_DEFAULT	ADC native bit resolution

8.5.2 Opening ADC Channel File

After the ADC driver is opened and initialized as described in Section 8.5.1, "Opening ADC Device," the channel driver file can be opened as "<device>:<channel_number>". Again, an initialization record is passed to the open call to initialize the ADC channel.

The following is an example for the MCF52xx family of microcontrollers as it can be found in the appropriate example code (see the */mqx/example/adc/adc.c* file).

```
static LWEVENT_STRUCT evn;

const ADC_INIT_CHANNEL_STRUCT adc_channel_param1 = {
        ADC_SOURCE_AN1,         /* physical ADC channel */
        ADC_CHANNEL_MEASURE_ONCE | ADC_CHANNEL_START_NOW,
                                /* one sequence is sampled after fopen */
        10,                     /* number of samples in one run sequence */
        100000,                 /* time offset from trigger point in us */
        500000,                 /* period in us (=500ms) */
        0,                      /* reserved - not used */
        10,                     /* circular buffer size (sample count) */
        ADC_TRIGGER_2,          /* logical trigger ID that starts this ADC channel */
        &evn                    /* pointer to event */
        0x01                    /* event mask to be set */
}
f = fopen("adc:temperature",(const char*)&adc_channel_param1);
```

ADC_TRIGGER_n and HW specific triggers are defined in *adc.h* and *adc_<CPU_name>.h*

The period time can be set just as a multiplication of the base period for devices using the PDB triggering. The base period can be set either by the IOCTL command or when opening the first channel (*period* parameter of the initialization structure).

The table below describes constants and flags you can pass in the initialization record when you call **fopen()** for the ADC channel device. They are defined in *adc.h* and *adc_<CPU_name>.h*.

Value	Description
"source" member of ADC_INIT_CHANNEL_STRUCT	
ADC_SOURCE_ANn	Physical ADC channel linked to the channel device file.
"flags" member of ADC_INIT_CHANNEL_STRUCT	
ADC_CHANNEL_MEASURE_LOOP	Measurement runs continuously. The **lwevent** is set periodically after each sampling sequence. The length of the sequence is specified in the *number_samples* member of ADC_INIT_CHANNEL_STRUCT. This flag is mutually exclusive with ADC_CHANNEL_MEASURE_ONCE.

Value	Description
ADC_CHANNEL_MEASURE_ONCE	One sequence is sampled. The length of the sequence is specified in the *number_samples* member of ADC_INIT_CHANNEL_STRUCT). This flag is mutually exclusive with ADC_CHANNEL_MEASURE_LOOP.
ADC_CHANNEL_START_TRIGGERED	Measurement starts after trigger is fired or after using the IOCTL_ADC_RUN_CHANNEL ioctl command. This flag is mutually exclusive with ADC_CHANNEL_START_NOW.
ADC_CHANNEL_START_NOW	Measurement starts immediately after fopen(). initiating with the IOCTL_ADC_RUN_CHANNEL ioctl command. This flag is mutually exclusive with ADC_CHANNEL_START_TRIGGERED.
ADC_CHANNEL_ACCUMULATE	Accumulate all samples from one sequence into one value.
“trigger” member of ADC_INIT_CHANNEL_STRUCT	
ADC_TRIGGER_n	Set of triggers assigned to the current channel file. ADC channel reacts to any of registered triggers. Multiple channels may be triggered by using IOCTL_ADC_FIRE_TRIGGER **ioctl** command.

8.6 Using IOCTL Commands

This section describes the I/O control commands that you use when you call **_io_ioctl()** for a particular ADC device driver. They are defined in *adc.h*.

IOCTL_ADC_xxx commands are deprecated. Use ADC_IOCTL_xxx naming convention as described in the following table.

Command	Description
ADC_IOCTL_RUN_CHANNEL	Initiates measurement sequence on the specified channel file.
ADC_IOCTL_RUN_CHANNELS or ADC_IOCTL_FIRE_TRIGGER	Fires one or more triggers. The trigger mask is passed directly to **ioctl** call as an argument.
ADC_IOCTL_STOP_CHANNEL	Stops measurement on specified channel file. No parameter is used.
ADC_IOCTL_STOP_CHANNELS	Stops measurement on all channels assigned to given set of triggers. The trigger mask is passed directly to **ioctl** call as an argument.
ADC_IOCTL_PAUSE_CHANNEL	Pauses measurement on specified channel file.

Command	Description
ADC_IOCTL_PAUSE_CHANNELS	Pauses measurement on all channels assigned to given set of triggers. The trigger mask is passed directly to **ioctl** call as an argument.
ADC_IOCTL_RESUME_CHANNEL	Resumes (after pausing) measurement on specified channel file.
ADC_IOCTL_RESUME_CHANNELS	Resumes (after pausing) measurement on all channels assigned to a given set of triggers. The trigger mask is passed directly to **ioctl** call as an argument.

8.6.1 Hardware-Specific IOCTL Commands

Hardware-specific commands are used to handle specific MCU behavior and hardware performance. These commands are not portable to other processor.

The following table summarizes MCF51EM, MCF51MM, and Kinetis family processor specific IOCTL commands.

Command	Description
ADC_IOCTL_CALIBRATE	Starts calibration process on a device. Command fails if any channel on a deviceis opened.
ADC_IOCTL_SET_CALIBRATION	Copies calibration data to the registers. A structure of type MCF51EM_ADC16_CALIB_STRUCT_PTR is passed as a parameter to the command. Command cannot be performed on channel file.
ADC_IOCTL_GET_CALIBRATION	Copies calibrated registers values to a calibration structure of type MCF51EM_ADC16_CALIB_STRUCT_PTR, which is passed as a parameter to the command. Command cannot be performed on channel file.
ADC_IOCTL_SET_LONG_SAMPLE	Sets long sampling time. For more information, see ADLSMP bit in MCU Reference Manual. Number of ADC periods (2, 6, 12 or 20) is passed as a parameter to the command. Command cannot be performed on channel file.
ADC_IOCTL_SET_SHORT_SAMPLE	Sets short sampling time. For more information, see ADLSMP bit in MCU Reference Manual. Command does not require a parameter. Command cannot be performed on channel file.
ADC_IOCTL_SET_HIGH_SPEED	Sets high speed conversion. For more information, see ADHSC bit in MCU Reference Manual. No parameter is passed to the command. Command does not require a parameter. Command cannot be performed on channel file.

Command	Description
ADC_IOCTL_SET_LOW_SPEED	Sets high speed conversion. For more information, see ADHSC bit in MCU Reference Manual. Command does not require a parameter. Command cannot be performed on channel file.
ADC_IOCTL_SET_HW_AVERAGIN G	Sets averaging. For more information, see AVGE bit in MCU Reference Manual. Number of samples used for averaging (0, 4, 8, 16, 32) is passed to the command as parameter. Value of zero disables averaging functionality. Command cannot be performed on channel file.
ADC_IOCTL_SET_IDELAY_PROCE SS	Controls the AD result value acquisition for a channel to be performed in IDELAY interrupt. Command does not require a parameter. Command cannot be performed on device file.
ADC_IOCTL_SET_INT_PROCESS	Controls the ADC result value acquisition for a channel to be performed in ADC interrupt. Command does not require a parameter. Command cannot be performed on device file.
ADC_IOCTL_SET_OFFSET	Sets the offset for ADC. For more information, see ADCOFS register in MCU Reference Manual. The value for the register is passed as a parameter. Command cannot be performed on channel file.
ADC_IOCTL_SET_PLUS_GAIN	Sets the plus gain for ADC. For more information, see ADCPG register in MCU Reference Manual. The value for the register is passed as a parameter. Command cannot be performed on channel file.
ADC_IOCTL_SET_MINUS_GAIN	Sets the minus gain for ADC. For more information, see ADCMG register in MCU Reference Manual. The value for the register is passed as a parameter. Command cannot be performed on channel file.
ADC_IOCTL_SET_IDELAY	Sets the IDELAY register with a value corresponding to a value passed as a parameter to the command and representing time in microseconds.
ADC_IOCTL_SET_IDELAYREG	Similar to ADC_IOCTL_SET_IDELAY, however, the parameter passed to the command is the raw value of IDELAY register.
ADC_IOCTL_SET_IDELAY_FCN	Sets application callback function of type PDB_INT_FCN for 'PDB idelay' ISR. The function pointer is passed as a parameter to the command.
ADC_IOCTL_SET_ERROR_FCN	Sets application callback function of type PDB_INT_FCN for 'PDB error' ISR. The function pointer is passed as a parameter to the command. This command cannot be run on MCF51MM.
ADC_IOCTL_SET_BASE_PERIOD	Sets period of PDB peripheral. The parameter passed to the command is the period time in microseconds.

Command	Description
ADC_IOCTL_TRIM_BASE_PERIOD	Similar to ADC_IOCTL_SET_BASE_PERIOD, however, the parameter passed to the command is the raw value of MOD register.
ADC_IOCTL_SET_DELAYREG	Sets the delay register for a channel. The parameter passed to the command is the raw value of DELAY register. Command cannot be performed on device file.
ADC_IOCTL_SET_TRIGGER	Sets the PDB block trigger source register for a channel. The parameter passed to the command is one of the ADC_PDB_TRIGSEL enum type.
ADC_IOCTL_SET_REFERENCE	Sets the reference voltage for ADC converter. The parameter passed to the is one of the ADC_REFERENCE enum type. Command cannot be performed on channel file.

NOTE

The PDB_INT_FCN is defined as:

```
typedef void (_CODE_PTR_ PDB_INT_FCN)(void);
```

The following table summarizes Kinetis-only processor specific IOCTL commands:

Command	Description
ADC_IOCTL_SET_PGA_GAIN	Sets GAIN of PGA. Use ADC_PGA_GAIN enum as a parameter. Can be applied only on channels that are amplified with PGA.
ADC_IOCTL_SET_PGA_GAIN	Gets GAIN of PGA as ADC_PGA_GAIN type. Can be applied only on channels that are amplified with PGA.
ADC_IOCTL_ENABLE_CHOPPING	Enables chopping. For more information, see the PGA Chapter in the MCU Reference Manual. Can be applied only on channels that are amplified with PGA.
ADC_IOCTL_DISABLE_CHOPPING	Disables chopping. For more information, see the PGA Chapter in the Reference Manual. Can be applied only on channels that are amplified with PGA.

The following table summarizes MCF51JE and MCF51MM specific IOCTL commands:

Command	Description
ADC_IOCTL_PIN_DISABLE	Disable ADC functionality despite ADC channel is opened. Convenient command to allow usage of another PIN alternative with lower priority than ADC such as GPIO.
ADC_IOCTL_PIN_ENABLE	Enable ADC functionality on given PIN. This command should be used for re-enabling ADC functionality temporarily disabled by ADC_IOCTL_PIN_DISABLE.

8.7 Example

For basic use, see MQX examples — ADC example in directory *mqx\examples\adc*.

8.8 Error Codes

Error code	Description
ADC_ERROR_ALLOC	Memory allocation error.
ADC_ERROR_ISR	Interrupt vector installation error.
ADC_ERROR_PARAM	Missing parameter.
ADC_ERROR_OPENED	File already opened.
ADC_ERROR_MISSING_DEVICE	Device was not opened prior to channel opening.
ADC_ERROR_BAD_PARAM	Bad parameter.
ADC_ERROR_FULL	Cannot open more files.
ADC_ERROR_NONEMPTY	Cannot run command if channel is still opened.
ADC_ERROR_ONLY_DEVICE	Cannot run command on channel file.
ADC_ERROR_ONLY_CHANNEL	Cannot run command on device file.

Hardware-specific errors for MCF51EM and MCF51MM processors:

Error code	Description
ADC_ERROR_PERIOD	Cannot run command when base period was not set.
ADC_ERROR_HWTRIGGER	Only HW trigger is supported.

Chapter 9 SPI Drivers

9.1 Overview

This chapter describes the SPI driver framework which provides a common interface for various SPI modules currently supporting DSPI module.

9.2 Location of Source Code

The source code for SPI drivers are located in source\io\spi.

9.3 Header Files

To use a SPI device driver, include the header files *spi.h* and device-specific *spi_xxxx.h* from *source\io\spi* in your application or in the BSP file *bsp.h*.

The files *spi_xxxx_prv.h* and *spi_prv.h* contain private definitions and data structures that SPI device drivers use. These files are required to compile an SPI device driver. There is no need to include these files directly in your application.

9.4 Internal Design of SPI Drivers

The SPI driver framework features layered design with two distinct layers: low level drivers and high level driver. The low level drivers are device specific and implement necessary hardware abstraction function sets. On the other hand, the high level driver is device independent and provides POSIX I/O adaptation on top of a low level driver including handling of concurrent access to the SPI bus from multiple tasks.

9.5 Installing SPI Driver

The SPI driver framework provides common function **_io_spi_install()** that either the BSP or the application calls.

The installation function calls low level driver initialization to configure appropriate pins for SPI, allocates memory necessary for keeping device state, and then calls **_io_dev_install()** internally to register a corresponding device in the IO subsystem.

The following is an example of an installation of the SPI device driver:

```
#if BSPCFG_ENABLE_SPI0
   _io_spi_install("spi0:", &_bsp_spi0_init);
#endif
```

This code can be found typically can in */mqx/bsp/init_bsp.c* file.

9.5.1 Initialization Record

The installation function requires a pointer to an initialization record to be passed to it. This record is used to initialize the device and the driver itself. Besides other information, the record contains a pointer to a device interface structure determining the low level driver to be used and pointer to its initialization data. The description of the initialization record and related data structures follows.

Main initialization record

```
typedef struct spi_init_struct
{
    SPI_DEVIF_STRUCT_CPTR DEVIF;

    const void _PTR_      DEVIF_INIT;

    SPI_PARAM_STRUCT      PARAMS;

    SPI_CS_CALLBACK       CS_CALLBACK;

    pointer               CS_USERDATA;

} SPI_INIT_STRUCT, _PTR_ SPI_INIT_STRUCT_PTR;
```

Parameters

DEVIF - Pointer to device interface structure defined by particular low level driver to be used

DEVF_INIT - Pointer to initialization data specific to the low level driver

PARAMS - Default transfer parameters to be used for newly opened file handles

CS_CALLBACK - Function implementing chip select control in software (not mandatory)

CS_USERDATA - Context data passed to chip select callback function (not mandatory)

Transfer parameters record

```
typedef struct spi_param_struct
{
    uint_32 BAUDRATE;

    uint_32 MODE;

    uint_32 FRAMESIZE;

    uint_32 CS;

    uint_32 ATTR;

    uint_32 DUMMY_PATTERN;

} SPI_PARAM_STRUCT, _PTR_ SPI_PARAM_STRUCT_PTR;
```

Parameters

BAUDRATE - Baud rate to use

MODE - Transfer mode (clock polarity and phase)

FRAMESIZE - Size of single SPI frame in bits

CS - Mask of chip select signals to use. No chip select signal is used if zero is specified.

ATTR - Additional attributes which may be used to enable a low level device specific functionality

DUMMY_PATTERN - Pattern to be shifted out to the bus during half-duplex read operation

Example of initialization records for DSPI (kinetis family, pxs20 and pxs30)

```
static const DSPI_INIT_STRUCT _bsp_dspi0_init = {
    0,                            /* SPI channel */
    CM_CLOCK_SOURCE_BUS           /* Relevant module clock source */
};

const SPI_INIT_STRUCT _bsp_spi0_init = {
    &_spi_dspi_devif,             /* Low level driver interface */
    &_bsp_dspi0_init,             /* Low level driver init data */
    {                             /* Default parameters: */
        10000000,                     /* Baudrate */
        SPI_CLK_POL_PHA_MODE0,        /* Mode */
        8,                            /* Frame size */
        1,                            /* Chip select */
        DSPI_ATTR_USE_ISR,            /* Attributes */
        0xFFFFFFFF                    /* Dummy pattern */
    }
};
```

9.6 Using the Driver

A file handle to the SPI device is obtained by *_io_open()* API call. Chip select mask may be optionally specified after colon character as file name part of the open string. Please note that specifying a zero chip select mask instructs the driver to use no chip select signals at all.

```
spifd = fopen("spi2:1", NULL); /* CS0 on bus spi2*/
```

The file handle obtains default transfer parameters defined in the initialization structure upon opening, including the chip select mask, unless it is specified in the open string. The transfer parameters may be changed later on using *_io_ioctl()* call. The transfer parameters are specific for particular file handle, that is, if multiple file handles are opened, each handle keeps its own set of transfer parameters.

Upon calling to *_io_read()* or *_io_write()*, the bus is first reserved for the file handle specified. If the bus is already reserved for another file handle, the call blocks wait until the bus is available to be reserved. After successful reservation of the bus, the SPI interface is configured according to the transfer parameters kept by the file handle, chip select signals are asserted and the requested amount of data is transferred (unless an error occurs). The bus then remains reserved and the chip select signals are kept asserted enabling further blocks of data to be transferred with continuously asserted chip select signals.

Read and write operation are strictly synchronous. The calling task is always blocked until read or write operation is complete or an error occurs.

The chip select signals are de-asserted and the bus is released by execution of either IO_IOCTL_FLUSH_OUTPUT or IO_IOCTL_SPI_FLUSH_DEASSERT_CS command or by closing the file handle with *_io_close()* call.

The flush operation does not need to wait for any buffers to be empty or any background operations to finish as there are none. It is solely used to finish the sequence of IO operations by de-asserting chip select and releasing the bus so that it may be accessed through other file handles.

As described above, the SPI driver may be concurrently used from multiple tasks using multiple file handles without needing any additional locking or synchronization in the application since the bus reservation mechanism, internal to the SPI driver, prevents collisions in multitasking environment.

9.7 Duplex Operation

The SPI driver is also capable of full-duplex operation in two different ways:

The first option is to use an extension of *_io_read()* operation. Since SPI bus itself is designed for full-duplex operation, the SPI driver has to shift out some data to the bus even if performing a read operation. Standard behavior of *_io_read()* is to act as half-duplex for the application, shifting out the dummy pattern previously set by IO_IOCTL_SPI_SET_DUMMY_PATTERN. To enable the full-duplex extension, a special flag SPI_FLAG_FULL_DUPLEX has to be either passed to *_io_open()*, or later on set using IO_IOCTL_SPI_SET_FLAGS. Once the flag is set, the *_io_read()* will shift out the content of the buffer passed to it while overwriting it with data being received, i.e. duplex operation on a single buffer is performed.

```
char buffer[11];

strcpy (buffer, "ABCDEFGHIJ");

/* ABCDEFGHIJ will be shifted to the bus and overwritten with data received */
read (spifd, buffer, 10);

fflush (spifd); /* chip select de-asserted */
```

The second option is to use IOCTL command IO_IOCTL_SPI_READ_WRITE which provides a true full-duplex operation by using distinct receive and transmit buffer. A parameter to this IOCTL command is SPI_READ_WRITE_STRUCT structure containing pointers to buffers and length of the transfer. Behavior of IO_IOCTL_SPI_READ_WRITE is not affected by a state of the SPI_FLAG_FULL_DUPLEX flag.

```
SPI_READ_WRITE_STRUCT rw;

rw.BUFFER_LENGTH = 10;
rw.WRITE_BUFFER = (char_ptr)send_buffer;
rw.READ_BUFFER = (char_ptr)recv_buffer;

if (SPI_OK == ioctl (spifd, IO_IOCTL_SPI_READ_WRITE, &rw)) /*chip select
asserted*/
{
  printf ("OK\n");
} else {
  printf ("ERROR\n");
}

fflush (spifd); /* chip select de-asserted */
```

9.8 Chip Selects Implemented in Software

SPI driver provides a way to implement chip select signals in software which is especially useful in the following scenarios:

- The application requires more CS signals than is supported by hardware.
- The hardware CS signals are multiplexed with another peripheral required for the application and thus cannot be used.
- External de-multiplexor or an I/O expander is to be used for CS signals.

A single callback function for CS handling may be registered per SPI device. The callback function registration is performed by the IO_IOCTL_SPI_SET_CS_CALLBACK IOCTL command. The parameter of the command is SPI_CS_CALLBACK_STRUCT which contains a pointer to the callback function and a pointer to the arbitrary context data for the callback function.

SPI driver than calls the function any time when a change of the CS signals state is necessary. Besides the context data, the function is also passed a desired state of the CS signals. The callback function is then responsible for changing the state of the CS by any method (e.g. using LWGPIO).

Please note that setting the callback function possibly affects all file handles associated with the same SPI device since the function is called for any change to the state of CS signals, regardless of the file handle used for operation which is causing the CS state change.

9.9 I/O Open Flags

This section describes the flag values which may be passed to *_io_fopen()*. Definitions of the flags may be found in *spi.h*.

Flag	Description
SPI_FLAG_HALF_DUPLEX or NULL	Read operation on file handle will behave the standard POSIX I/O way.
SPI_FLAG_FULL_DUPLEX	Enables extension to standard POSIX I/O for full-duplex operation.
SPI_FLAG_NO_DEASSERT_ON_FLUSH	If set, call to fflush() or IO_IOCTL_FLUSH_OUTPUT command causes bus to be released without de-asserting CS signals. However the state of CS signals may then change as a result of a transfer performed using any file handle associated with the same SPI device.

9.10 I/O Control Commands

This section describes the I/O control commands defined by the SPI driver to be used with *_io_ioctl()* call.

The common commands are defined in *spi.h*. The commands are used to get or set parameters operating on the given file handle only and do not affect other file handles associated with the same SPI device, unless stated otherwise. Please note that low level driver does not necessarily have to support all combinations of the transfer parameters. If the selected combination is not supported, the read/write operations on the file handle will fail returning IO_ERROR.

Command	Description	Parameter
IO_IOCTL_SPI_GET_BAUD	Gets the BAUD rate.	uint_32_ptr
IO_IOCTL_SPI_SET_BAUD	Sets the baud rate. A supported baud rate closest to the given one will be used.	uint_32_ptr
IO_IOCTL_SPI_GET_MODE	Gets clock polarity and phase mode.	uint_32_ptr
IO_IOCTL_SPI_SET_MODE	Sets clock polarity and phase mode.	uint_32_ptr
IO_IOCTL_SPI_GET_DUMMY_PATTERN	Gets dummy pattern for half-duplex read.	uint_32_ptr
IO_IOCTL_SPI_SET_DUMMY_PATTERN	Sets dummy pattern for half-duplex read.	uint_32_ptr
IO_IOCTL_SPI_GET_TRANSFER_MODE	Gets operation mode (master/slave).	uint_32_ptr
IO_IOCTL_SPI_SET_TRANSFER_MODE	Sets operation mode (master/slave).	uint_32_ptr
IO_IOCTL_SPI_GET_ENDIAN	Gets endian mode of the transfer.	uint_32_ptr
IO_IOCTL_SPI_SET_ENDIAN	Sets endian mode of the transfer.	uint_32_ptr
IO_IOCTL_SPI_GET_FLAGS	Gets open flags.	uint_32_ptr
IO_IOCTL_SPI_SET_FLAGS	Sets open flags.	uint_32_ptr
IO_IOCTL_SPI_GET_STATS	Gets communication statistics (structure defined in *spi.h*).	SPI_STATISTICS_STRUCT_PTR
IO_IOCTL_SPI_CLEAR_STATS	Clears communication statistics.	ignored
IO_IOCTL_FLUSH_OUTPUT	Releases the bus and de-asserts CS signals unless SPI_FLAG_NO_DEASSERT_ON_FLUSH flag is set.	ignored
IO_IOCTL_SPI_FLUSH_DEASSERT_CS	Releases the bus and de-asserts CS.	ignored
IO_IOCTL_SPI_GET_FRAMESIZE	Gets number of bits of single SPI frame.	uint_32_ptr
IO_IOCTL_SPI_SET_FRAMESIZE	Sets number of bits of single SPI frame.	uint_32_ptr
IO_IOCTL_SPI_GET_CS	Gets chip select mask.	uint_32_ptr
IO_IOCTL_SPI_SET_CS	Sets chip select mask.	uint_32_ptr
IO_IOCTL_SPI_SET_CS_CALLBACK	Sets callback function for handling CS state changes in software. Setting CS callback function possibly affects all file handles associated with the same SPI device.	SPI_CS_CALLBACK_STRUCT_PTR
IO_IOCTL_SPI_READ_WRITE	Performs simultaneous write and read full duplex operation.	SPI_READ_WRITE_STRUCT_PTR

Commands which are not handled by the high level driver are passed to the low level driver. Such device specific IOCTLs may be implemented by the low level driver to enable access to special capabilities of the hardware.

9.11 Clock Modes

Clock mode values passed to *_io_ioctl()* with the IO_IOCTL_SPI_SET_MODE command:

Signal	Description
SPI_CLK_POL_PHA_MODE0	Clock signal inactive low and bit sampled on rising edge.
SPI_CLK_POL_PHA_MODE1	Clock signal inactive low and bit sampled on falling edge.
SPI_CLK_POL_PHA_MODE2	Clock signal inactive high and bit sampled on falling edge.
SPI_CLK_POL_PHA_MODE3	Clock signal inactive high and bit sampled on rising edge.

9.12 Transfer Modes

Transfer mode values passed to *_io_ioctl()* with the IO_IOCTL_SPI_SET_TRANSFER_MODE command:

Signal	Description
SPI_DEVICE_MASTER_MODE	Master mode (generates clock).
SPI_DEVICE_SLAVE_MODE	Slave mode.

9.13 Endian Mode

Endian mode values passed to *_io_ioctl()* with the IO_IOCTL_SPI_SET_ENDIAN command:

Signal	Description
SPI_DEVICE_BIG_ENDIAN	Big endian most significant bit transmitted first.
SPI_DEVICE_LITTLE_ENDIAN	Little endian least significant bit transmitted first.

9.14 Error Codes

Following the SPI, specific error codes are defined:

Error Code	Description
SPI_ERROR_MODE_INVALID	Given clock mode is unknown or not supported.
SPI_ERROR_TRANSFER_MODE_INVALID	Given transfer mode is unknown or unsupported.
SPI_ERROR_BAUD_RATE_INVALID	Given baud rate cannot be used.
SPI_ERROR_ENDIAN_INVALID	Given endian mode is unknown or unsupported.
SPI_ERROR_CHANNEL_INVALID	Attempt to access non-existing SPI channel.
SPI_ERROR_DEINIT_FAILED	Driver de-initialization failed.
SPI_ERROR_INVALID_PARAMETER	Given parameter is invalid.
SPI_ERROR_FRAMESIZE_INVALID	Frame size not supported.

9.15 Compatibility and Migration Guide

This chapter contains comparison of the new SPI driver framework and the legacy SPI driver, and describes possible issues related to migration of an application to the new SPI driver framework.

Resource sharing

The most significant change in the new SPI driver framework is the support of the concurrent bus access from multiple tasks.

The legacy SPI driver does not provide any special support for concurrent access. There may be only a single file handle associated with the SPI device and the application has ensure not using the file handle from multiple tasks at the same time. When there are more slaves connected to the bus, the application also has to handle changing the chip select and possibly also other transfer parameters by issuing IOCTL commands whenever communication with a different slave is going to take place.

By contrast, the new SPI driver framework fully supports resource sharing. There may be simultaneously open multiple file handles associated with the same SPI device. Each file handle keeps its own set of transfer parameters. In a typical use case, each file handle represents a virtual communication channel connected to a slave device. The application then simply performs read/write/flush operations on the file handles and the SPI driver framework takes care about bus sharing and automatic reconfiguration of transfer parameters any time when this is necessary.

Backward compatibility

If there is only a single file handle associated with the SPI device open, the behavior from the application's end is pretty much the same as the one of the polled legacy SPI driver. Most of the applications designed to be used with polled legacy SPI driver should work without any significant changes with the new SPI driver framework.

This is even true if the application is using the file handle for communication with multiple slaves. However, it is strongly recommended to adapt such application to the new SPI driver framework, that is, pass the responsibility for bus sharing and transfer parameter switching to the driver by using multiple file handles.

Error reporting

The principle of operation of the new driver framework implies that transfer parameters which are set by using an IOCTL command cannot be applied immediately since this might affect the data transfer in progress performed using another file handle. Because of this, the transfer parameters are only checked for sanity in the IOCTL call, but no error is reported if a transfer parameter or their combination is not supported by the low level driver. Such condition is reported later on upon attempt to perform read/write operation by returning IO_ERROR and setting task error variable to a specific error code.

Chip select callback

The legacy SPI driver provides a possibility to register a callback function for each chip select signal which is to be handled by software. Because this approach brought unnecessary complexity and overhead, the new SPI driver framework supports only a single chip select callback function which takes an additional parameter specifying desired state of chip select signals. The callback function then handles switching of particular signals internally. Unlike transfer parameters, the chip select callback applies to the SPI device as whole, that is, it is shared by all file handles.

Transfer statistics

The new SPI driver framework uses a different data structure for transfer statistics than did the legacy SPI driver. Therefore, the IOCTL command, IO_IOCTL_SPI_GET_STATS, is not backward compatible. The statistics data are kept separately for each file handle. Only generic transfer statistics are provided. No low level specific events are counted. Since the support for statistics is not compiled in by default, it is necessary to enable this feature by defining BSPCFG_ENABLE_SPI_STATS to 1 if desired.

Dummy pattern support

Since the SPI bus itself is designed for full-duplex operation, the SPI driver has to shift out some data on the bus even if performing read operation. The new SPI driver framework provides a possibility to set a dummy pattern to be shifted out during the read operation. See 9.7, “Duplex Operation" for details.

Low level initialization

The new SPI driver framework performs the low level initialization when _io_spi_install() is called, so that the SPI bus is brought to a defined state at the time of installation of the device.

By contrast, the installation of the legacy SPI driver does not touch the hardware at all. The low level initialization is performed when the device is open, that is, the state of the SPI bus with the legacy driver is not defined until the SPI device is used for the first time.

Chapter 10 Legacy SPI Drivers

10.1 Overview

This chapter describes the legacy SPI driver which is considered obsolete. The legacy SPI drivers will be replaced by the new SPI driver framework in the future. See Chapter 9, “SPI Drivers".

SPI device driver is a common interface for various SPI modules currently supporting ColdFire V1 SPI16 and QSPI. The driver includes:

- SPI interrupt-driven I/O - Available for all types of SPI modules
- SPI polled I/O - Available for all types of SPI modules

10.2 Location of Source Code

Driver	Location
SPI interrupt-driven	`source\io\spi\int`
SPI polled	`source\io\spi\polled`

10.3 Header Files

To use an SPI device driver, include the header files *spi.h* and device-specific *spi_xxxx.h* from *source\io\spi* in your application or in the BSP file *bsp.h*. Use the header files according to the following table.

Driver	Header file
SPI interrupt-driven	`spi.h`
SPI polled	`spi.h`

The files *spi_mcf5xxx_xxxx_prv.h*, *spi_pol_prv.h*, and *spi_int_prv.h* contain private data structures that SPI device driver uses. You must include these files if you recompile an SPI device driver. You may also want to look at the file as you debug your application.

10.4 Installing Drivers

Each SPI device driver provides an installation function that either the BSP or the application calls. The function then calls **_io_dev_install()** internally. Different installation functions exist for different SPI hardware modules. Please see the BSP initialization code in *init_bsp.c* for functions suitable for your hardware (*xxxx* in the function names below). Installation function configures appropriate pins to SPI functionality and initializes driver according to initialization record.

Driver	Installation function
Interrupt-driven	• _xxxx_qspi_int_install() • _xxxx_spi16_int_install()
Polled	• _xxxx_qspi_polled_install() • _xxxx_spi16_polled_install()

Example of installing the QSPI device driver:

```
#if BSPCFG_ENABLE_SPI0
   _mcf5xxx_qspi_polled_install("spi0:", &_bsp_qspi0_init);
#endif
```

This code can be found typically can in */mqx/bsp/init_bsp.c* file.

10.4.1 Initialization Record

Each installation function requires a pointer to the initialization record to be passed to it. This record is used to initialize the device and software when the device is opened for the first time. The record is unique to each possible device and the fields required along with initialization values are defined in the device-specific header files.

Synopsis for QSPI (mcf52259)

```
#include <spi_mcf5xxx_qspi.h>
typedef struct mcf5xxx_qspi_init_struct
{
 uint_32 CHANNEL;
 uint_32 CS;
 uint_32 TRANSFER_MODE;
 uint_32 BAUD_RATE;
 uint_32 CLOCK_SPEED;
 uint_32 CLOCK_POL_PHASE;
 uint_32 RX_BUFFER_SIZE;
 uint_32 TX_BUFFER_SIZE;
} MCF5XXX_QSPI_INIT_STRUCT, _PTR_ MCF5XXX_QSPI_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - SPI channel to initialize.

CS - Default chip selected for use unless changed by IOCTL command.

TRANSFER_MODE - SPI transfer mode (SPI_DEVICE_MASTER_MODE or SPI_DEVICE_SLAVE_MODE).

BAUD_RATE - Desired baud rate.

CLOCK_SPEED - Clock speed used by the SPI module to calculate baud rate.

CLOCK_POL_PHASE - Clock polarity and phase (SPI_CLK_POL_PHA_MODEx).

RX_BUFFER_SIZE - Maximum size of each receive.

TX_BUFFER_SIZE - Maximum size of each transmit.

Synopsis for mcf51xx - SPI8 and SPI16 (example for SPI8)

```
#include <spi_spi16.h>
typedef struct _spi16_init_struct
{
 uint_32 CHANNEL;
 uint_32 CS;
 uint_32 CLOCK_SPEED;
 uint_32 BAUD_RATE;
 uint_32 RX_BUFFER_SIZE;
 uint_32 TX_BUFFER_SIZE;
 uint_32 VECTOR;
 uint_32 TRANSFER_MODE;
 uint_32 CLOCK_POL_PHASE;
} SPI16_INIT_STRUCT, _PTR_ SPI16_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - SPI channel to initialize.

CS - Default chip selected for use unless changed by IOCTL command.

CLOCK_SPEED - Clock speed used by the SPI module (used to calculate baud rate).

BAUD_RATE - Desired baud rate.

RX_BUFFER_SIZE - Maximum size of each receive.

TX_BUFFER_SIZE - Maximum size of each transmit.

VECTOR - Interrupt vector to use if interrupt driven.

TRANSFER_MODE - SPI transfer mode (SPI_DEVICE_MASTER_MODE or SPI_DEVICE_SLAVE_MODE).

CLOCK_POL_PHASE - Clock polarity and phase (SPI_CLK_POL_PHA_MODEx).

10.5 Driver Services

The SPI serial device driver provides these services:

API	Calls	
	Interrupt-driven	Polled
_io_fopen()	_io_spi_int_open()	_io_spi_polled_open()
_io_fclose()	_io_spi_int_close()	_io_spi_polled_close()
_io_read()	_io_spi_int_read()	_io_spi_polled_read()
_io_write()	_io_spi_int_write()	_io_spi_polled_write()
_io_ioctl()	_io_spi_int_ioctl()	_io_spi_polled_ioctl()

Read/write operations automatically activate CS signals according to the previous setting via IO_IOCTL_SPI_SET_CS command.

10.6 I/O Open Flags

This section describes the flag values you can pass when you call *_io_fopen()* for a particular interrupt-driven or polled SPI device driver. They are defined in *spi.h*.

Flag	Description
SPI_FLAG_HALF_DUPLEX or NULL	Sets the communication in both directions, but only one direction at a time (not simultaneously).
SPI_FLAG_FULL_DUPLEX	Sets the communication in both directions simultaneously. **Note:** Not applicable when using single-wire (BIO) mode.
SPI_FLAG_NO_DEASSERT_ON_FLUSH	No CS signals are deactivated during call to fflush() or IO_IOCTL_FLUSH_OUTPUT command.

10.7 I/O Control Commands

This section describes the I/O control commands that you use when you call **_io_ioctl()** for a particular interrupt-driven or polled SPI device driver. These commands are available for both interrupt-driven and polled SPI device driver. However, some of these commands are not applicable for particular SPI hardware modules. The commands are defined in *spi.h*.

Command	Description
IO_IOCTL_SPI_GET_BAUD	Gets the BAUD rate.
IO_IOCTL_SPI_SET_BAUD	Sets the BAUD rate (finds closest to the given one).
IO_IOCTL_SPI_GET_MODE	Gets clock polarity and sample mode.
IO_IOCTL_SPI_SET_MODE	Sets clock polarity and sample mode.
IO_IOCTL_SPI_ENABLE_MODF	Enables mode fault detection in master mode, and automatic switch to the slave mode.
IO_IOCTL_SPI_DISABLE_MODF	Disables master mode fault detection.
IO_IOCTL_SPI_GET_TRANSFER_MODE	Gets operation mode.
IO_IOCTL_SPI_SET_TRANSFER_MODE	Sets operation mode.
IO_IOCTL_SPI_GET_ENDIAN	Gets endian transfer mode.
IO_IOCTL_SPI_SET_ENDIAN	Sets endian transfer mode.
IO_IOCTL_SPI_DEVICE_ENABLE	Enables SPI device.
IO_IOCTL_SPI_DEVICE_DISABLE	Disables SPI device.
IO_IOCTL_SPI_GET_FLAGS	Gets duplex mode flags.
IO_IOCTL_SPI_SET_FLAGS	Sets duplex mode flags.

Command	Description
IO_IOCTL_SPI_GET_STATS	Gets communication statistics (structure defined in *spi.h*).
IO_IOCTL_SPI_CLEAR_STATS	Clears communication statistics
IO_IOCTL_FLUSH_OUTPUT	Waits for transfer to finish, deactivate CS signals only if opening flag SPI_FLAG_NO_DEASSERT_ON_FLUSH was not set.
IO_IOCTL_SPI_FLUSH_DEASSERT_CS	Waits for transfer to finish and always deactivate CS signals regardless on opening flags.
IO_IOCTL_SPI_GET_FRAMESIZE	Gets number of bits per one transfer.
IO_IOCTL_SPI_SET_FRAMESIZE	Sets number of bits per one transfer.
IO_IOCTL_SPI_GET_CS	Gets chip select enable mask.
IO_IOCTL_SPI_SET_CS	Sets chip select enable mask.
IO_IOCTL_SPI_SET_CS_CALLBACK	Sets callback function to handle chip select assertion and deassertion. Chip select is automatically asserted during write(), read(), and IO_IOCTL_SPI_READ_WRITE. Callback function may use any method how to control CS signal e.g. using GPIO driver. This functionality is available only ColdFire V1 SPI device driver. QSPI and DSPI controls CS signal automatically,
IO_IOCTL_SPI_READ_WRITE	Performs simultaneous write and read full duplex operation. Parameter of this IO control command is a pointer to SPI_READ_WRITE_STRUCT structure, where READ_BUFFER, WRITE_BUFFER pointers and BUFFER_LEN has to be provided.
IO_IOCTL_SPI_KEEP_QSPI_CS_ACTIVE	Applies only for QSPI HW module. Modifies QSPI HW chip selects behaviour. Default value is TRUE. If TRUE, transfers longer than 16 frames are possible with CS asserted until flush() is called - with a side effect of holding all chip selects low between transfers (HW limitation). If FALSE, the longest continuous transfer (CS asserted) is 16 frames. Read/write requests above 16 frames are automatically divided into continuous transfers of 16 frames (and the rest). CS is automatically deasserted after each transfer. Furthermore, in interrupt mode, CS is asserted/deasserted for each frame. This is because HW FIFO is not used for compatibility reasons with other SPI modules that don't use queue.

10.8 Example

This example shows simultaneous read/write operation. Send and receive buffers have to point to memory of BUFFER_LENGTH size. One buffer can be used for both WRITE_BUFFER and READ_BUFFER.

```
SPI_READ_WRITE_STRUCT rw;

rw.BUFFER_LENGTH = 10;
rw.WRITE_BUFFER = (char_ptr)send_buffer;
rw.READ_BUFFER = (char_ptr)recv_buffer;
printf ("READ WRITE ... ");

if (SPI_OK == ioctl (spifd, IO_IOCTL_SPI_READ_WRITE, &rw)) /*chip select asserted*/
{
  printf ("OK\n");
} else {
  printf ("ERROR\n");
}

fflush (spifd); /* chip select de-asserted */
printf ("Simultaneous write and read - EEPROM read from 0x%08x (%d):\n",
        SPI_EEPROM_ADDR1, rw.BUFFER_LENGTH);
```

10.9 Clock Modes

This section describes the clock mode values you can pass when you call **_io_ioctl()** with the IO_IOCTL_SPI_SET_MODE command. They are defined in *spi.h*.

Signal	Description
SPI_CLK_POL_PHA_MODE0	Clock signal inactive low and bit sampled on rising edge.
SPI_CLK_POL_PHA_MODE1	Clock signal inactive low and bit sampled on falling edge.
SPI_CLK_POL_PHA_MODE2	Clock signal inactive high and bit sampled on falling edge.
SPI_CLK_POL_PHA_MODE3	Clock signal inactive high and bit sampled on rising edge.

10.10 Transfer Modes

This section describes the operation mode values you can pass when you call **_io_ioctl()** with the IO_IOCTL_SPI_SET_TRANSFER_MODE command. They are defined in *spi.h*.

Signal	Description
SPI_DEVICE_MASTER_MODE	Master mode (generates clock).
SPI_DEVICE_SLAVE_MODE	Slave mode.
SPI_DEVICE_BIO_MASTER_MODE	Master mode using single-wire bidirectional transfer.
SPI_DEVICE_BIO_SLAVE_MODE	Slave mode using single-wire bidirectional transfer.

10.11 Endian Transfer Modes

This section describes the endian transfer mode values you can pass when you call **_io_ioctl()** with the IO_IOCTL_SPI_SET_ENDIAN command. They are defined in *spi.h*.

Signal	Description
SPI_DEVICE_BIG_ENDIAN	Big endian, most significant bit transmitted first.
SPI_DEVICE_LITTLE_ENDIAN	Little endian, least significant bit transmitted first.

10.12 Duplex Mode Flags

This section describes the flag values you can pass when you call **_io_ioctl()** with the IO_IOCTL_SPI_SET_FLAGS command. They are defined in *spi.h*.

Flag	Description
SPI_FLAG_HALF_DUPLEX	Sets communication in both directions, but only one direction at a time.
SPI_FLAG_FULL_DUPLEX	Sets communication in both directions simultaneously. **Note:** Not applicable when using single-wire (BIO) mode.
SPI_FLAG_NO_DEASSERT_ON_FLUSH	No CS signals are deactivated during call to fflush() or IO_IOCTL_FLUSH_OUTPUT command.

10.13 Error Codes

No additional error codes are generated.

Error Code	Description
SPI_ERROR_MODE_INVALID	Given clock mode is unknown.
SPI_ERROR_TRANSFER_MODE_INVALID	Given transfer mode is unknown.
SPI_ERROR_BAUD_RATE_INVALID	Given baud rate is zero.
SPI_ERROR_ENDIAN_INVALID	Given endian mode is unknown.
SPI_ERROR_CHANNEL_INVALID	Opening non-existing SPI channel.
SPI_ERROR_DEINIT_FAILED	Closing driver failed.
SPI_ERROR_INVALID_PARAMETER	Given parameter is invalid (NULL).

Chapter 11 I²C Driver

11.1 Overview

This chapter describes I²C device driver. The driver includes:

- I²C interrupt-driven I/O
- I²C polled I/O

11.2 Source Code Location

Driver	Location
I²C interrupt-driven	`source\io\i2c\int`
I²C polled	`source\io\i2c\polled`

11.3 Header Files

To use an I²C device driver, include the header files, *i2c.h,* and device-specific, *i2c_mcfxxxx.h,* from *source\io\i2c* in your application or in the BSP file *bsp.h*. Use the header files according to the following table.

Driver	Header file
I²C interrupt-driven	• `i2c.h` • `i2c_mcfxxxx.h`
I²C polled	• `i2c.h` • `i2c_mcfxxxx.h`

The files *i2c_mcfxxxx_prv.h, i2c_pol_prv.h,* and *i2c_int_prv.h* contain private data structures that I²C device driver uses. You must include these files if you recompile an I²C device driver. You may also want to look at the file as you debug your application.

11.4 Installing Drivers

Each I²C device driver provides an installation function that either the BSP or the application calls. The function then calls **_io_dev_install()** internally. Different installation functions exist for different I²C hardware modules. Please see the BSP initialization code in *init_bsp.c* for functions suitable for your hardware (mcf*xxxx* in the function names below).

Driver	Installation function
Interrupt-driven	_mcfxxxx_i2c_int_install()
Polled	_mcfxxxx_i2c_polled_install()

11.4.1 Initialization Records

Each installation function requires a pointer to the initialization record to be passed to it. This record is used to initialize the device and software when the device is opened for the first time. The record is unique to each possible device and the fields required along with initialization values are defined in the device-specific header files.

Synopsis for kinetis family, mcf51jf and mcf51qm

```
#include <i2c_ki2c.h>
typedef struct ki2c_init_struct
{
uint_8         CHANNEL;
uint_8         MODE;
uint_8         STATION_ADDRESS;
uint_32        BAUD_RATE;
#if !(BSP_TWRMCF51FD || BSP_TWRMCF51JF || BSP_TWRMCF51QM)
_int_level     LEVEL;
_int_priority  SUBLEVEL;
#endif
uint_32        TX_BUFFER_SIZE;
uint_32        RX_BUFFER_SIZE;
}KI2C_INIT_STRUCT, _PTR_ KI2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE).

STATION_ADDRESS - I2C station address for the channel (slave mode).

BAUD_RATE - Desired baud rate.

LEVEL - Interrupt level to use if interrupt driven (Kinetis only).

SUBLEVEL - Sub level within the interrupt level to use if interrupt driven (Kinetis only).

TX_BUFFER_SIZE - Tx buffer size (interrupt driven only).

RX_BUFFER_SIZE - Rx buffer size (interrupt driven only).

Synopsis for mcf51XX family (except of mcf51jf and mcf51qm)

```
#include <i2c_mcf51xx.h>
typedef struct mcf51xx_i2c_init_struct
{
uint_8  CHANNEL;
uint_8  MODE;
uint_8  STATION_ADDRESS;
uint_32 BAUD_RATE;
```

```
uint_32 TX_BUFFER_SIZE;
uint_32 RX_BUFFER_SIZE;
} MCF51XX_I2C_INIT_STRUCT, _PTR_ MCF51XX_I2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE).

STATION_ADDRESS - I2C station address for the channel (slave mode).

BAUD_RATE - Desired baud rate.

TX_BUFFER_SIZE - Tx buffer size (interrupt driven only).

RX_BUFFER_SIZE - Rx buffer size (interrupt driven only).

Synopsis for mcf52XX

```
#include <i2c_mcf52xx.h>
typedef struct mcf52xx_i2c_init_struct
{
uint_8        CHANNEL;
uint_8        MODE;
uint_8        STATION_ADDRESS;
uint_32       BAUD_RATE;
_int_level    LEVEL;
_int_priority SUBLEVEL;
uint_32       TX_BUFFER_SIZE;
uint_32       RX_BUFFER_SIZE;
} MCF52XX_I2C_INIT_STRUCT, _PTR_ MCF52XX_I2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE).

STATION_ADDRESS - I2C station address for the channel (slave mode).

BAUD_RATE - Desired baud rate.

LEVEL - Interrupt level to use if interrupt driven.

SUBLEVEL - Sub level within the interrupt level to use if interrupt driven.

TX_BUFFER_SIZE - Tx buffer size (interrupt driven only).

RX_BUFFER_SIZE - Rx buffer size (interrupt driven only).

Synopsis for mcf53XX and mcf54XX (example for mcf53XX)

```
#include <i2c_mcf53xx.h>
typedef struct mcf53xx_i2c_init_struct
{
uint_8      CHANNEL;
uint_8      MODE;
```

```
uint_8      STATION_ADDRESS;
uint_32     BAUD_RATE;
_int_level  LEVEL;
uint_32     TX_BUFFER_SIZE;
uint_32     RX_BUFFER_SIZE;
} MCF53XX_I2C_INIT_STRUCT, _PTR_ MCF53XX_I2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE).

STATION_ADDRESS - I2C station address for the channel (slave mode).

BAUD_RATE - Desired baud rate.

LEVEL - Interrupt level to use if interrupt driven.

TX_BUFFER_SIZE - Tx buffer size (interrupt driven only).

RX_BUFFER_SIZE - Rx buffer size (interrupt driven only).

Synopsis for pxs30

```
#include <i2c_qi2c.h>
typedef struct qi2c_init_struct
{
uint_8  CHANNEL;
uint_8  MODE;
uint_8  STATION_ADDRESS;
uint_32 BAUD_RATE;
uint_32 LEVEL;
uint_32 TX_BUFFER_SIZE;
uint_32 RX_BUFFER_SIZE;
} QI2C_INIT_STRUCT, _PTR_ QI2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE).

STATION_ADDRESS - I2C station address for the channel (slave mode).

BAUD_RATE - Desired baud rate.

LEVEL - Interrupt level to use if interrupt driven.

TX_BUFFER_SIZE - Tx buffer size (interrupt driven only).

RX_BUFFER_SIZE - Rx buffer size (interrupt driven only).

Synopsis for mpc5125

```
#include <i2c_mpc512x.h>
typedef struct mpc512x_i2c_init_struct
{
uint_8  CHANNEL;
```

```
uint_8  MODE;
uint_8  STATION_ADDRESS;
uint_32 BAUD_RATE;
uint_32 LEVEL;
uint_32 TX_BUFFER_SIZE;
uint_32 RX_BUFFER_SIZE;
} MPC512X_I2C_INIT_STRUCT, _PTR_ MPC512X_I2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE).

STATION_ADDRESS - I2C station address for the channel (slave mode).

BAUD_RATE - Desired baud rate.

LEVEL - Interrupt level to use if interrupt driven.

TX_BUFFER_SIZE - Tx buffer size (interrupt driven only).

RX_BUFFER_SIZE - Rx buffer size (interrupt driven only).

Synopsis for mpc8308

```
#include <i2c_mpc83xx.h>
typedef struct mpc83xx_i2c_init_struct
{
uint_8  CHANNEL;
uint_8  MODE;
uint_8  STATION_ADDRESS;
uint_32 BAUD_RATE;
uint_32 LEVEL;
uint_32 SUBLEVEL;
uint_32 TX_BUFFER_SIZE;
uint_32 RX_BUFFER_SIZE;
} MPC83xx_I2C_INIT_STRUCT, _PTR_ MPC83xx_I2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE).

STATION_ADDRESS - I2C station address for the channel (slave mode).

BAUD_RATE - Desired baud rate.

LEVEL - Interrupt level to use if interrupt driven.

SUBLEVEL - Sub level within the interrupt level to use if interrupt driven.

TX_BUFFER_SIZE - Tx buffer size (interrupt driven only).

RX_BUFFER_SIZE - Rx buffer size (interrupt driven only).

Example

The following code is an example for the MCF52xx microcontrollers family as it can be found in the appropriate BSP code. See, for example, the *init_i2c0.c* file.

```
const MCF52XX_I2C_INIT_STRUCT _bsp_i2c0_init = {
   0,                      /* I2C channel    */
   BSP_I2C0_MODE,          /* I2C mode       */
   BSP_I2C0_ADDRESS,       /* I2C address    */
   BSP_I2C0_BAUD_RATE,     /* I2C baud rate  */
   BSP_I2C0_INT_LEVEL,     /* I2C int level  */
   BSP_I2C0_INT_SUBLEVEL,  /* I2C int sublvl */
   BSP_I2C0_TX_BUFFER_SIZE,/* I2C int tx buf */
   BSP_I2C0_RX_BUFFER_SIZE /* I2C int rx buf */
};
```

11.5 Driver Services

The I^{2}C serial device driver provides these services:

API	Calls	
	Interrupt-driven	Polled
_io_fopen()	_io_i2c_int_open()	_io_i2c_polled_open()
_io_fclose()	_io_i2c_int_close()	_io_i2c_polled_close()
_io_read()	_io_i2c_int_read()	_io_i2c_polled_read()
_io_write()	_io_i2c_int_write()	_io_i2c_polled_write()
_io_ioctl()	_io_i2c_int_ioctl()	_io_i2c_polled_ioctl()

11.6 I/O Control Commands

This section describes the I/O control commands used when you call **_io_ioctl()** for a particular interrupt-driven or polled I^{2}C driver. They are defined in *i2c.h*.

Command	Description
IO_IOCTL_I2C_SET_BAUD	Sets the baud rate.
IO_IOCTL_I2C_GET_BAUD	Gets the baud rate.
IO_IOCTL_I2C_SET_MASTER_MODE	Sets device to the I^{2}C master mode.
IO_IOCTL_I2C_SET_SLAVE_MODE	Sets device to the I^{2}C slave mode
IO_IOCTL_I2C_GET_MODE	Gets the mode previously set.
IO_IOCTL_I2C_SET_STATION_ADDRESS	Sets the device's I^{2}C slave address.
IO_IOCTL_I2C_GET_STATION_ADDRESS	Gets the device's I^{2}C slave address.
IO_IOCTL_I2C_SET_DESTINATION_ADDRESS	Sets the address of the called device (master only).
IO_IOCTL_I2C_GET_DESTINATION_ADDRESS	Gets the address of the called device (master only).

Command	Description
IO_IOCTL_I2C_SET_RX_REQUEST	Sets a number of bytes in advance to read before stop.
IO_IOCTL_I2C_REPEATED_START	Initiates I^2C repeated start condition (master only).
IO_IOCTL_I2C_STOP	Generates I^2C stop condition (master only).
IO_IOCTL_I2C_GET_STATE	Gets the actual state of transmission.
IO_IOCTL_I2C_GET_STATISTICS	Gets the communication statistics (structure defined in *i2c.h*.)
IO_IOCTL_I2C_CLEAR_STATISTICS	Clears the communication statistics.
IO_IOCTL_I2C_DISABLE_DEVICE	Disables I^2C device.
IO_IOCTL_I2C_ENABLE_DEVICE	Enables I^2C device.
IO_IOCTL_FLUSH_OUTPUT	Flushes the output buffer, waits for the transfer to finish.
IO_IOCTL_I2C_GET_BUS_AVAILABILITY	Gets the actual bus state (idle/busy).

11.7 Device States

This section describes the device state values you can get when you call **_io_ioctl()** with the IO_IOCTL_I2C_GET_STATE command. They are defined in *i2c.h*.

State	Description
I2C_STATE_READY	Ready to generate start condition (master) and transmission.
I2C_STATE_REPEATED_START	Ready to initiate repeated start (master) and transmission.
I2C_STATE_TRANSMIT	Transmit in progress.
I2C_STATE_RECEIVE	Receive in progress.
I2C_STATE_ADDRESSED_AS_SLAVE_RX	Device addressed by another master to receive.
I2C_STATE_ADDRESSED_AS_SLAVE_TX	Device addressed by another master to transmit.
I2C_STATE_LOST_ARBITRATION	Device lost arbitration. It doesn't participate on the bus anymore.
I2C_STATE_FINISHED	Transmit interrupted by NACK, or all requested bytes received.

11.8 Device Modes

This section describes the device state values you can get when you call **_io_ioctl()** with the IO_IOCTL_I2C_GET_MODE command. They are defined in *i2c.h*.

Mode	Description
I2C_MODE_MASTER	I^2C master mode, generates clock, start/rep.start/stop conditions, and sends address.
I2C_MODE_SLAVE	I^2C slave mode, reacts when its station address is being sent on the bus.

11.9 Bus Availability

This section describes the bus states you can get when you call **_io_ioctl()** with the IO_IOCTL_I2C_GET_BUS_AVAILABILITY command. They are defined in *i2c.h*.

Bus State	Description
I2C_BUS_IDLE	Stop condition occurred. No i2c transmission on the bus.
I2C_BUS_BUSY	Start/Repeated started detected. Transmission in progress.

11.10 Error Codes

No additional error codes are generated.

Error code	Description
I2C_OK	Operation successful.
I2C_ERROR_DEVICE_BUSY	Device is currently working.
I2C_ERROR_CHANNEL_INVALID	Wrong init data.
I2C_ERROR_INVALID_PARAMETER	Invalid parameter passed (NULL).

Chapter 12 FlashX Driver

12.1 Overview

This section contains information about NOR Flash device drivers that accompany the Freescale MQX.

12.2 Source Code Location

The source code for flash drivers resides in *source\io\flashx*.

12.3 Header Files

To use flash drivers, include *flashx.h* and device-specific header file (for example *flash_ftfl.h*) in your application or in the BSP file *bsp.h*.

The files with **prv.h* postfix contain private constants and data structures that flash drivers use.

12.4 Hardware Supported

MQX FlashX driver enables reading and writing on-chip flash memory for all devices supported by the Freescale MQX. Additionally, it supports some of the external flash memory types. See sub-directories in the *mqx/source/io/flashx* driver directory.

12.5 Driver Services

Flash drivers provide the following full set of services.

API	Calls
_io_fopen()	_io_flashx_open()
_io_fclose()	_io_flashx_close()
_io_read()	_io_flashx_read()
_io_write()	_io_flashx_write()
_io_ioctl()	_io_flashx_ioctl()

12.6 Installing Drivers

A flash driver provides installation function that either the BSP or the application calls. The function in turn calls *_io_dev_install_ext* internally.

12.7 Installing and Uninstalling Flash Devices

To install a driver for a generic flash device, call **_io_flashx_install()**.

This function initializes the generic driver.

12.7.1 _io_flashx_install

Synopsis

```
_max_uint _io_flashx_install(char_ptr id, FLASHX_INIT_STRUCT _PTR_ init_ptr)
```

Parameters

- *id* [in] — String identifying the NOR Flash controller device for **fopen()**.
- *init_ptr* [in] — Structure containing initialization information for the flashx driver.

12.7.2 _io_flashx_uninstall

Synopsis

```
_max_uint _io_flashx_uinstall()
```

12.7.3 FLASHX_INIT_STRUCT

Synopsis

```
struct flashx_init_struct {
  _mem_size                             BASE_ADDR;
  const FLASHX_BLOCK_INFO_STRUCT _PTR_  HW_BLOCK;
  const FLASHX_FILE_BLOCK _PTR_         FILE_BLOCK;
  const FLASHX_DEVICE_IF_STRUCT _PTR_   DEVICE_IF;
  _mqx_uint                             WIDTH;
  _mqx_uint                             DEVICES;
  _mqx_uint                             WRITE_VERIFY;
  pointer                               DEVICE_SPECIFIC_INIT;
} FLASHX_INIT_STRUCT, _PTR_ FLASHX_INIT_STRUCT_PTR;
```

Parameters

- BASE_ADDR [IN] — Base address of the device.
- HW_BLOCK [IN] — Array of HW blocks describing the organization of Flash memory.
- FILE_BLOCK [IN] — Array of BSP predefined files that can be opened with the Flash driver.
- DEVICE_IF [IN] — Array of device interface which includes functions that map functionality.
- WIDTH [IN] — The bus data lines used for external devices.
  — 1 (accessed by bytes)
  — 2 (accessed by words)
  — 4 (accessed as long words)
  — 8 (accessed as double longs)
- WRITE_VERIFY [IN] — If true, a comparison of the original data and the flash write is made.

- DEVICE_SPECIFIC_INIT [IN] — If required by the low level driver, user can pass information from the BSP.

12.7.4 FLASHX_BLOCK_INFO_STRUCT

This structure contains information about the flash structure: sector size in one block, number of blocks with the same sector size, and offset of the block from the start of the flash address space. An array of the structures used in the FLASHX_INIT_STRUCT forms device block map. The block map for specific flash devices can be found in mqx\source\io\flashx\<producer_name>\<device_name>.c file. The blocks do not have to follow each other, so a space between blocks is acceptable. It is required that the blocks do not intersect and that they are listed in ascending order in the array by their starting address.

Synopsis

```
struct flashx_block_info_struct {
  _mqx_uint        NUM_SECTORS;
  _mem_size        START_ADDR;
  _mem_size        SECT_SIZE;
  uint_32          SPECIAL_TAG;
} FLASHX_BLOCK_INFO_STRUCT, _PTR_ FLASHX_BLOCK_INFO_STRUCT;
```

Parameters

- NUM_SECTORS [IN] — Number of sectors of identical size.
- START_ADDR [IN] — Starting address (offset) of this block of sectors - this address is relative to the base address passed to FlashX driver installation routine. The physical address of the block can be computed as: *BASE_ADDR + START_ADDR*.
- SECT_SIZE [IN] — Size of the sectors in this block.
- SPECIAL_TAG [IN] — Additional information if required by low-level driver.

Example of block info structures for AT49BV1614 flash memory (with various sector size):

```
#define AT49BV1614A_SECTOR_SIZE_1 (0x2000)
#define AT49BV1614A_SECTOR_SIZE_2 (0x10000)
#define AT49BV1614A_NUM_SECTORS_1 (8)
#define AT49BV1614A_NUM_SECTORS_2 (31)

FLASHX_BLOCK_INFO_STRUCT _at49bv1614a_block_map_16bit[] =  {
  { AT49BV1614A_NUM_SECTORS_1, 0, AT49BV1614A_SECTOR_SIZE_1, 0}, /* 8x8KB */
  { AT49BV1614A_NUM_SECTORS_2, 0x10000, AT49BV1614A_SECTOR_SIZE_2, 0},/* 31x64KB */
  { 0, 0, 0, 0 } /* zero block terminates the array */
};
```

12.7.5 FLASHX_FILE_BLOCK

Every installed instance of the Flash driver needs to have a specified set of files that can be opened with the driver. The files are enumerated in the array of the **FLASHX_FILE_BLOCK** in the **init_flashx.c** file of the BSP directory. The array is passed to the Flash driver initialization structure as a parameter.

Synopsis

```
typedef struct flashx_file_block
```

```
{
    char_ptr const   FILENAME;
    _mem_size        START_ADDR;
    _mem_size        END_ADDR;
} FLASHX_FILE_BLOCK, _PTR_ FLASHX_FILE_BLOCK_PTR;
```

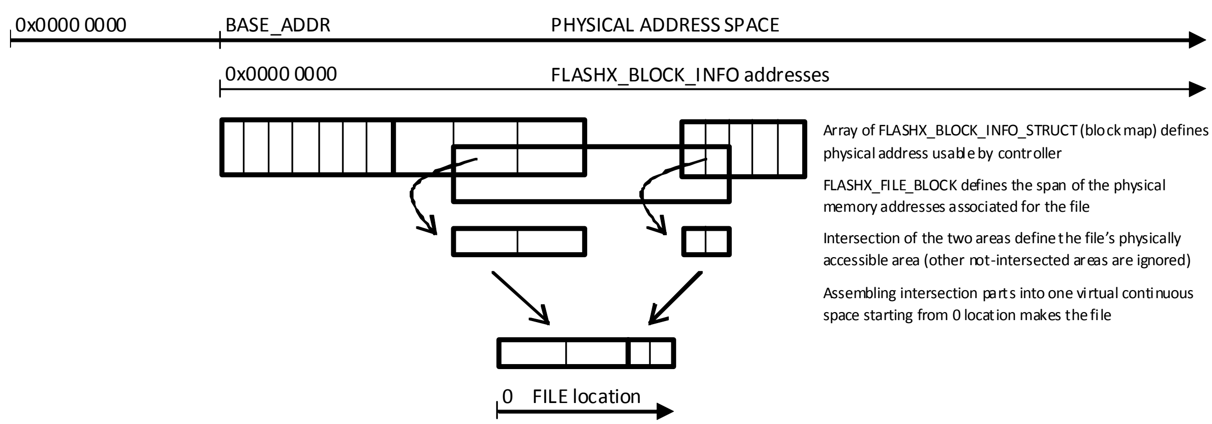
Parameters

- FILENAME[IN] — Name of the file that can be opened by the Flash driver.
- START_ADDR [IN] — Starting, physical, address of the beginning of the file.
- END_ADDR [IN] — Ending, last, address of the byte comprised by the file.

If the starting physical address falls to a block mapped into the physical address space, then it must correspond to the physical address of the first byte of a sector.

If the ending physical address falls to a block mapped into the physical address space, then it must correspond to the physical address of the last byte of a sector.

The file represents virtual continuous area and makes a, sort of, an abstraction for application developer. The intersection between file's area defined by **START_ADDR** and **END_ADDR** member of **FLASHX_FILE_BLOCK** structure and the Flash block array defined by **FLASHX_BLOCK_INFO_STRUCT** forms a set of Flash sectors which can be accessed linearly through the file as shown in the image below.



12.8 I/O Control Commands

This section describes the I/O control commands that you use when you call **_io_ioctl()**. The commands apply to all flash drivers except if stated otherwise. They are defined in *flash.h*.

Command	Description	Parameters
FLASH_IOCTL_GET_BASE_ADDRESS	Returns the base address of the flash memory.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_NUM_SECTORS	Returns the number of sectors in the flash file.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_SECTOR_BASE	Returns the start address of the current sector - sector at current file location.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_SECTOR_SIZE	Returns the size of the current sector - sector at current file location.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_WIDTH	Returns the width of the flash device.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_BLOCK_GROUPS	Returns the number of blocks in the device block map.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_BLOCK_MAP	Returns the address of the device block map description defined by an array of FLASHX_BLOCK_INFO_STRUCT.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_FLUSH_BUFFER	Writes out all cached sectors if there is valid data in the cache.	none (NULL)
FLASH_IOCTL_ENABLE_BUFFERING	Enables write-back caching of the single Flash sector. This **ioctl** can only be enabled if FLASH_IOCTL_ENABLE_SECTOR_CACHE is enabled.	none (NULL)
FLASH_IOCTL_DISABLE_BUFFERING	Disables write back cache.	none (NULL)
FLASH_IOCTL_ERASE_SECTOR	Erases the sector at a current file location.	none (NULL)
FLASH_IOCTL_ERASE_CHIP	Erases the entire flash device.	none (NULL)
FLASH_IOCTL_ENABLE_SECTOR_CACHE	Enables allocating the sector buffer. This affects write-back caching. Also, it restricts some driver functionality. See FLASH_IOCTL_DISABLE_SECTOR_CACHE).	none (NULL)

Command	Description	Parameters
FLASH_IOCTL_DISABLE_SECTOR_CACHE	Disables allocating the sector in the memory. Intention of this feature is RAM saving, the sector size could be large enough to decrease performance of the application, but it restricts driver functionality. Sector allocation is needed in the following cases: - Partial sector overwrite when the destination area is not blank. In these cases the sector allocation is **not** required: - Full sector write. - Partial sector overwrite when the destination area is blank. Disabling sector cache also rules out write-back caching. See FLASH_IOCTL_ENABLE_BUFFERING.	none (NULL)
IO_IOCTL_GET_NUM_SECTORS	Returns the number of sectors for MFS device. The default MSF_SECTOR_SIZE is 512 bytes.	*param_ptr* - pointer to 32b variable
IO_IOCTL_DEVICE_IDENTIFY	Returns to upper layer, what kind of device is it. It is a physical flash device, capable of being erased, read, and written. Flash devices are not interrupt driven, so IO_DEV_ATTR_POLL is included. Used in MFS driver.	*param_ptr* - pointer to block identification array, required by MFS
IO_IOCTL_GET_BLOCK_SIZE	Returns the fixed MFS sector size which is usually 512.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_SWAP_FLASH_AND _RESET	Swaps the flash memory blocks. Works only with the dual flash memory controllers.	none (NULL)
FLASH_IOCTL_WRITE_ERASE_CMD _FROM_FLASH_ENABLE	Sets up to run the low level flash write and erase routines from internal flash memory. Supported only on the dual flash memory controllers.	none (NULL)
FLASH_IOCTL_WRITE_ERASE_CMD _FROM_FLASH_DISABLE	Sets up to run the low level flash write and erase routines from RAM. Supported only on the dual flash memory controllers.	none (NULL)

The following table lists the FlexNVM specific IOCTL commands.

Command	Description	Parameters
FLEXNVM_IOCTL_READ_RESOURCE	The read resource command allows the user to read data from special-purpose memory.	*param_ptr* - pointer to struct FLEXNVM_READ_RSRC_STRUCT
FLEXNVM_IOCTL_SET_PARTITION_CODE	Set partition code and EEPROM size - change FlexNVM organization.	*param_ptr* - pointer to struct FLEXNVM_PROG_PART_STRUCT
FLEXNVM_IOCTL_GET_PARTITION_CODE	Read FlexNVM partition code.	*param_ptr* - pointer to FLEXNVM_PROG_PART_STRUCT structure which is filled by function
FLEXNVM_IOCTL_SET_FLEXRAM_FN	Enable FlexEEPROM mode in FlexNVM.	*param_ptr* - pointer to uint_8 - FlexRAM Function Control Code: 0xFF - FlexRAM available as RAM 0x00 - FlexRAM available for EEPROM
FLEXNVM_IOCTL_WAIT_EERDY	Wait until FlexEEPROM is ready after write operation.	none (NULL)
FLEXNVM_IOCTL_GET_EERDY	Get FlexEEPROM ready flag from FlexNVM controller. This flag provides information about readiness state of FlexNVM in EEPROM mode.	*param_ptr* - pointer to uint_32 - EEReady flag value: 0x1 - ready

12.9 Data Types Used with the FlexNVM

This section describes the data types used by the FlexNVM driver.

12.9.1 FLEXNVM_READ_RSRC_STRUCT

Synopsis:

```
typedef struct {
  uint_32 ADDR;
  uint_8  RSRC_CODE;
  uint_32 RD_DATA;
} FLEXNVM_READ_RSRC_STRUCT;
```

Parameters:

ADDR - flash address.

RSRC_CODE - resource selector.

RD_DATA - readed resources data.

12.9.2 FLEXNVM_PROG_PART_STRUCT

Synopsis:

```
typedef struct {
  uint_8 EE_DATA_SIZE_CODE;
  uint_8 FLEXNVM_PART_CODE;
} FLEXNVM_PROG_PART_STRUCT;
```

Parameters:

EE_DATA_SIZE_CODE - eeprom data size code which is composed of two parts - EE_SPLIT and EE_SIZE (FLEXNVM_EE_SPLIT_x_x | FLEXNVM_EE_SIZE_xxxx).

Configuration values for EE_SPLIT are:

- FLEXNVM_EE_SPLIT_1_7
- FLEXNVM_EE_SPLIT_1_3
- FLEXNVM_EE_SPLIT_1_1

Configuration values for EE_SIZE are:

- FLEXNVM_EE_SIZE_4096
- FLEXNVM_EE_SIZE_2048
- FLEXNVM_EE_SIZE_1024
- FLEXNVM_EE_SIZE_512
- FLEXNVM_EE_SIZE_256
- FLEXNVM_EE_SIZE_128
- FLEXNVM_EE_SIZE_64
- FLEXNVM_EE_SIZE_32
- FLEXNVM_EE_SIZE_0

FLEXNVM_PART_CODE - FlexNVM partition code. Possible values are:

- FLEXNVM_PART_CODE_DATA256_EE0
- FLEXNVM_PART_CODE_DATA224_EE32
- FLEXNVM_PART_CODE_DATA192_EE64
- FLEXNVM_PART_CODE_DATA128_EE128
- FLEXNVM_PART_CODE_DATA32_EE224
- FLEXNVM_PART_CODE_DATA64_EE192
- FLEXNVM_PART_CODE_DATA0_EE256
- FLEXNVM_PART_CODE_NOPART

12.10 Error Codes

Flash drivers only use the MQX I/O error codes.

Chapter 13 SD Card Driver

13.1 Overview

This section describes the SD Card driver that accompanies the MQX release. SD Card protocols up to version 2.0 (SDHC) are supported.

The driver uses block access with a block size of 512 bytes. The MFS file system can be installed on the top of this driver to implement FAT file access as shown on Figure 13-1.

Supported driver subfamilies:

- SD Card SPI driver — Transfers the data blocks via SPI Bus using polling mode of operation.
- SD Card ESDHC driver — Transfers the data blocks via SD Bus using ESDHC driver where available.

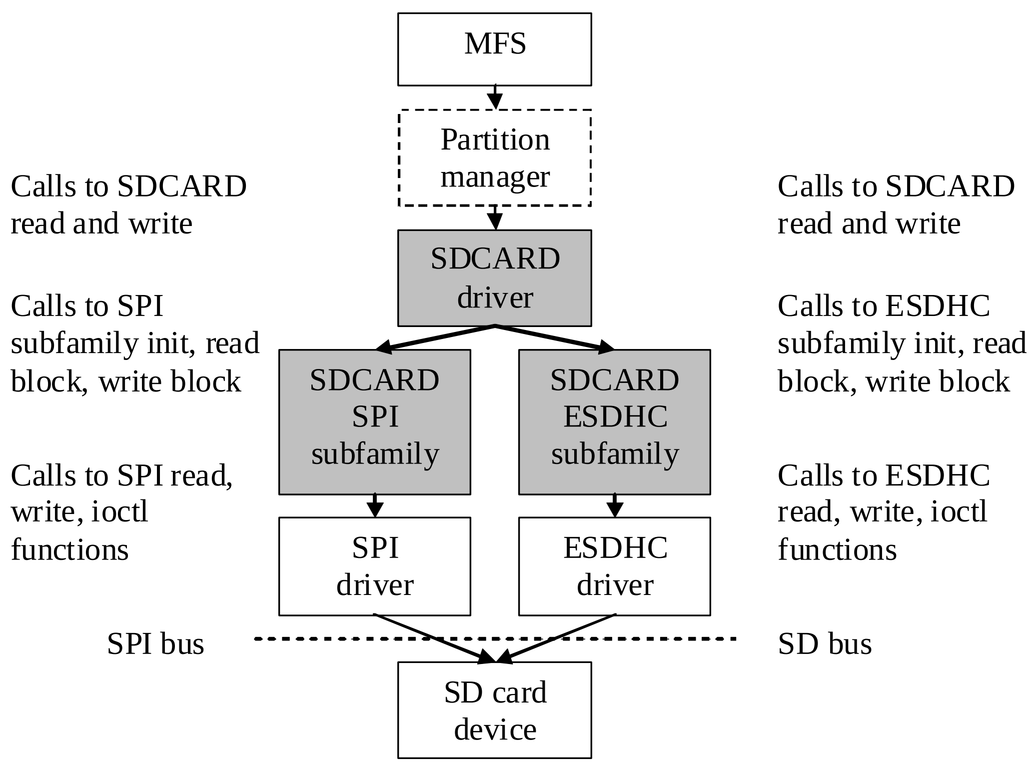


Figure 13-1. SD Card driver stack

13.2 Source Code Location

The source files for SD Card driver are located in `source\io\sdcard` directory.

13.3 Header Files

To use the SD Card driver, include the header file *sdcard.h* and a subfamily header file, for example *sdcard_spi.h*, in your application or in the BSP header file (*bsp.h*). The *sdcard_prv.h* file contains private constants and data structures used internally by the driver.

13.4 Installing Driver

The SD Card driver provides an installation function that the application may call. Installation function creates internal structures within MQX I/O subsystem and makes the driver available for public use. The parameters of installation function are:

- String identifier
- Pointer to the SD Card initialization structure
- A handle to the low-level communication device

The default initialization structure, *_bsp_sdcard0_init*, is created in the BSP, *init_sdcard0.c*, file. You can also define your own structure. Handle of low-level communication device should match the needs of the driver "subfamily" implementation. For SPI, a handle to open SPI device configured to half duplex mode should be passed.

```
_mqx_int _io_sdcard_install
   (
      /* [IN] A string that identifies the device for fopen */
      char_ptr               identifier,

      /* [IN] SD card initialization parameters */
      SDCARD_INIT_STRUCT_PTR init,

      /* [IN] Already opened communication descriptor */
      FILE_PTR               com_device
   )
```

SD Card is typically installed in the application code after opening a low-level communication device driver (SPI).

Read/Write protection and card presence detection is handled separately by using GPIO pins. BSP defines `BSP_SDCARD_GPIO_DETECT` and `BSP_SDCARD_GPIO_DETECT` pins for this purpose.

13.4.1 Initialization Record

The installation function requires a pointer to the initialization record to be passed to it. This record provides with abstraction of the communication channel used to interface the SDCARD.

Synopsis

```
#include <sdcard.h>
typedef struct sdcard_init_struct
{
boolean (_CODE_PTR_ INIT_FUNC)(MQX_FILE_PTR);
boolean (_CODE_PTR_ READ_FUNC)(MQX_FILE_PTR, uchar_ptr, uint_32);
```

```
boolean (_CODE_PTR_ WRITE_FUNC)(MQX_FILE_PTR, uchar_ptr, uint_32);
uint_32  SIGNALS;
} SDCARD_INIT_STRUCT, _PTR_ SDCARD_INIT_STRUCT_PTR;
```

Parameters for SPI interface

INIT_FUNC - initialization function, set to `_io_sdcard_spi_init`.

READ_FUNC - function to perform read operation, set to `_io_sdcard_spi_read_block`.

WRITE_FUNC - function to perform write operation, set to `_io_sdcard_spi_write_block`.

SIGNALS - determines SPI chip select for SDCARD communication.

Parameters for SDHC interface

INIT_FUNC - initialization function, set to `_io_sdcard_sdhc_init`.

READ_FUNC - function to perform read operation, set to `_io_sdcard_sdhc_read_block`.

WRITE_FUNC - function to perform write operation, set to `_io_sdcard_sdhc_write_block`.

SIGNALS - determines width of SDHC bus (SDHC_BUS_WIDTH_1 or SDHC_BUS_WIDTH_4).

Parameters for ESDHC interface

INIT_FUNC - initialization function, set to `_io_sdcard_esdhc_init`.

READ_FUNC - function to perform read operation, set to `_io_sdcard_esdhc_read_block`.

WRITE_FUNC - function to perform write operation, set to `_io_sdcard_esdhc_write_block`.

SIGNALS - determines width of SDHC bus (ESDHC_BUS_WIDTH_1, ESDHC_BUS_WIDTH_4 or ESDHC_BUS_WIDTH_8).

Example

The following code is found in the appropriate BSP code (*init_sdcard0.c*).

```
const SDCARD_INIT_STRUCT _bsp_sdcard0_init = {
        _io_sdcard_spi_init,
        _io_sdcard_spi_read_block,
        _io_sdcard_spi_write_block,
        BSP_SDCARD_SPI_CS
};
```

13.4.2 Driver Services

The SD Card device driver provides these services:

API	Calls	Description
`_io_fopen()`	`_io_sdcard_open()`	Calls the driver subfamily specific init function to set up low level communication and detect an initialize card and to get type and capacity of the card.
`_io_fclose()`	`_io_sdcard_close()`	`_io_fopen()` `_io_fclose()` just closes the SD Card driver. It doesn't affect the low-level communication device (which remains opened).
`_io_read()`	`_io_sdcard_read_blocks()`	`_io_read()` and `_io_write()` functions call appropriate subfamily specific functions for read block and write block.
`_io_write()`	`_io_sdcard_write_blocks()`	
`_io_ioctl()`	`_io_sdcard_ioctl()`	Used to get information about the driver/card capabilities.

13.5 I/O Control Commands

This section describes the I/O control commands that you use when you call `_io_ioctl()`. The commands are defined in *sdcard.h*.

Command	Description
`IO_IOCTL_GET_BLOCK_SIZE`	Returns the size of block in bytes. This ioctl command is mandatory for using a device with MFS.
`IO_IOCTL_GET_NUM_SECTORS`	Returns number of blocks available in the SD card.This ioctl command is mandatory for using a device with MFS.
`IO_IOCTL_DEVICE_IDENTIFY`	Returns flags describing SD card capabilities.This ioctl command is mandatory for using device with MFS.

13.6 Example

See example provided with the MQX installation located in: `mfs\examples\sdcard` directory.

Chapter 14 RTC Driver

14.1 Overview

This section describes the Real Time Clock (RTC) driver that accompanies the MQX release. This driver is a common interface for both RTC and Independent Real Time Clock (IRTC) peripheral modules.

The RTC driver implements custom API and does not follow the standard driver interface (I/O Subsystem).

14.2 Source Code Location

The source files for the RTC driver are located in `source\io\rtc` directory. The file prefix *rtc_* is used for all RTC module related API files and the file prefix *irtc_* is used for all IRTC module related API files.

14.3 Header Files

To use the RTC driver with the RTC peripheral module, include the header file named *rtc.h* and platform specific (*rtc_mcf52xx.h*) in your application or in the BSP header file (*bsp.h*).

To use the RTC driver with the IRTC peripheral module, include the device-specific header files *irtc_mcfxxxx.h* in your application or in the BSP header file (*bsp.h*).

For Kinetis platforms, include the header file *krtc.h* into in your application or in the BSP header file (*bsp.h*).

14.4 API Function Reference - RTC Module Related Functions

This sections serves as a function reference for the RTC module(s).

14.4.1 _rtc_init()

This function (re)initializes the RTC module.

Synopsis

```
uint_32 _rtc_init(uint_32 flags)
```

Parameters

flags [in] — A combination of initialization flags.

Description

The following initialization flags can be passed when the *_rtc_init()* function is called:

- RTC_INIT_FLAG_CLEAR - Clears RTC time, alarm, and stopwatch.
- RTC_INIT_FLAG_RESET - Disables and clears all interrupts and the stopwatch (even if cleared).
- RTC_INIT_FLAG_ENABLE - Installs HW interrupt and runs the RTC.

Return Value

- MQX_OK (success)

Example

The following example shows how to initialize the RTC module.

```
_rtc_init(RTC_INIT_FLAG_RESET | RTC_INIT_FLAG_ENABLE);
```

14.4.2 _rtc_isr()

This is the interrupt service routine for the RTC module.

Synopsis

```
void _rtc_isr(pointer ptr)
```

Parameters

ptr [in] — RTC module register structure pointer.

Description

This function serves as a template of the RTC module interrupt service routine. It is up to the user to implement the code for individual RTC interrupt types (alarm, stopwatch, time change).

Return Value

- None

14.4.3 _rtc_int_install()

This function installs the ISR for the RTC module.

Synopsis

```
uint_32 _rtc_int_install(pointer isr)
```

Parameters

isr [in] — pointer to user ISR code.

Description

This function installs the defined interrupt service routine for the RTC module. The modified _rtc_isr() function of the RTC driver can be registered, or you can write your own routine.

Return Value

- MQX_OK (Success)
- Other value if not successful

Example

The following example shows how to install user-defined ISR *my_rtc_isr()* for the RTC module.

```
printf ("Installing RTC interrupt... ");
if (MQX_OK != _rtc_int_install (my_rtc_isr))
{
        printf ("Error!\n");
}
```

14.4.4 _rtc_int_enable()

This function enables/disables RTC interrupts.

Synopsis

```
uint_32 _rtc_int_enable(
          boolean           enable,
          uint_32           bitmask)
```

Parameters

enable [in] — Enables or disable interrupts.

bitmask [in] — Bitmask of affected interrupts.

Description

This function enables/disables RTC interrupts based on the specified bitmask. The definition of the RTC interrupt request masks can be found in the device-specific header files.

Return Value

- Bitmask of the new interrupt enable state

Example

The following example shows how to disable all RTC interrupts.

```
_rtc_int_enable(FALSE, RTC_INT_ALL_MASK);
```

14.4.5 _rtc_clear_requests()

This function clears the RTC interrupt requests.

Synopsis

```
void _rtc_clear_requests(uint_32 bitmask)
```

Parameters

bitmask [in] — Bitmask of affected interrupts.

Description

This function clears RTC interrupts based on the specified bitmask. The definition of the RTC interrupt request masks can be found in the device-specific header files.

Return Value

- None

Example

The following example shows how to clear the RTC stopwatch interrupt.

```
_rtc_clear_requests (MCF54XX_RTC_ISR_SW);
```

14.4.6 _rtc_get_status()

This function returns the status of the RTC interrupt requests.

Synopsis

```
uint_32 _rtc_get_status(void)
```

Parameters

None

Description

This function returns bitmask of pending RTC interrupt requests. The definition of the RTC interrupt request masks can be found in the device-specific header files.

Return Value

- Bitmask of actual RTC interrupt requests + RTC enabled bit

14.4.7 _rtc_set_time()

This function sets the RTC time.

Synopsis

```
void _rtc_set_time(RTC_TIME_STRUCT_PTR time)
```

Parameters

time [in] — The time to be set as an RTC time.

Description

This function sets the RTC time according to the given time struct.

Return Value

- None

Example

The following example shows how to set the RTC time to 1.1.2010, 12:30.

```
RTC_TIME_STRUCT_PTR rtc_time

rtc_time->seconds   = 0;
rtc_time->minutes   = 30;
rtc_time->hours     = 12;
rtc_time->days      = 1;
rtc_time->month     = 1;
rtc_time->year      = 2010;
_rtc_set_time (&rtc_time);
```

14.4.8 _rtc_get_time()

This function returns the actual RTC time.

Synopsis

```
void _rtc_get_time(RTC_TIME_STRUCT_PTR time)
```

Parameters

time [out] — The actual RTC time.

Description

This function gets the actual RTC time and stores it in the given time struct.

Return Value

- None

14.4.9 _rtc_set_alarm()

This function sets the RTC alarm.

Synopsis

```
void _rtc_set_alarm(RTC_TIME_STRUCT_PTR time)
```

Parameters

time [in] — The time to be set as an RTC alarm time.

Description

This function sets the RTC alarm according to the given time struct.

Return Value

- None

Example

The following example shows how to set the RTC alarm time to 1.1.2010, 12:30.

```
RTC_TIME_STRUCT_PTR alarm_time

alarm_time->seconds   = 0;
alarm_time->minutes   = 30;
alarm_time->hours     = 12;
alarm_time->days      = 1;
alarm_time->month     = 1;
alarm_time->year      = 2010;
_rtc_set_alarm (&alarm_time);
```

14.4.10 _rtc_get_alarm()

This function returns the RTC alarm time.

Synopsis

```
_mqx_int _rtc_get_alarm(RTC_TIME_STRUCT_PTR time)
```

Parameters

time [out] — The RTC alarm time.

Description

This function gets the RTC alarm time and stores it in the given time struct.

Return Value

- None

14.4.11 _rtc_set_stopwatch()

This function sets the RTC stopwatch.

Synopsis

```
void _rtc_set_stopwatch(uint_32 minutes)
```

Parameters

minutes [in] — Number of minutes to count down.

Description

This function sets the RTC stopwatch decrementer value in minutes. Stopwatch decrements each new RTC minute and stops (disables) at -1. The stopwatch tolerance is +1 minute because decrementer changes its value each time the second counter rolls over 59 seconds.

Return Value

- None

Example

The following example shows how to set the RTC stopwatch to 5 minutes.

```
_rtc_set_stopwatch(5);
```

14.4.12 _rtc_get_stopwatch()

This function returns the actual value of the RTC stopwatch decrementer.

Synopsis

```
uint_32 _rtc_get_stopwatch(void)
```

Parameters

none

Description

This function returns the actual value of the RTC stopwatch decrementer.

Return Value

- The actual RTC minute stopwatch counter value

14.4.13 _rtc_time_to_mqx_time()

This function transforms RTC time format to the MQX time format.

Synopsis

```
void _rtc_time_to_mqx_time(
         RTC_TIME_STRUCT_PTR rtc_time,
         TIME_STRUCT_PTR     mqx_time)
```

Parameters

rtc_time [in] — RTC time representation.

mqx_time [out] — MQX time representation.

Description

This function transforms RTC time format to the MQX time format. RTC time range is wider (65536 days vs. 49710 days). Overflow is not checked and milliseconds are set to 0.

Return Value

- None

14.4.14 _rtc_time_from_mqx_time()

This function transforms the MQX time format to RTC time format.

Synopsis

```
void _rtc_time_from_mqx_time(
         TIME_STRUCT_PTR     mqx_time,
         RTC_TIME_STRUCT_PTR  rtc_time)
```

Parameters

mqx_time [in] — MQX time representation.

rtc_time [out] — RTC time representation.

Description

This function transforms the MQX time format to RTC time format. MQX time range is shorter (49710 days vs. 65536 days). Milliseconds are ignored.

Return Value

- None

14.4.15 _rtc_sync_with_mqx()

This function synchronizes RTC time with the MQX time.

Synopsis

```
void _rtc_sync_with_mqx(boolean update_mqx)
```

Parameters

update_mqx [in] — TRUE = sets the MQX time based on RTC time
FALSE = sets RTC time based on the MQX time

Description

This function allows setting the MQX time based on RTC time and vice versa.

Return Value

- MQX_OK
- RTC_INVALID_TIME, if entered date is out of MCU RTC registers range, for example, 1984 is the minimal year on mcf51mm.

14.4.16 _rtc_set_time_mqxd()

This function sets the RTC time.

Synopsis

```
_mqx_int _rtc_set_time_mqxd(DATE_STRUCT_PTR time)
```

Parameters

time [in] — The time to be set as an RTC time.

Description

This function sets the RTC time according to mqx DATE_STRUCT.

Return Value

- MQX_OK

Example

The following example shows how to set the RTC time to 1.1.2010, 12:30.

```
DATE_STRUCT rtc_time

rtc_time.MILLISEC = 0;
rtc_time.SECOND = 0;
rtc_time.MINUTE = 30;
rtc_time.HOUR = 12;
rtc_time.DAY = 1;
rtc_time.MONTH = 1;
rtc_time.YEAR = 2010;
_rtc_set_time_mqxd (&rtc_time);
```

14.4.17 _rtc_get_time_mqxd()

This function returns the actual RTC time.

Synopsis

```
void _rtc_get_time_mqxd(DATE_STRUCT_PTR time)
```

Parameters

time [in] — The actual RTC time.

Description

This function gets the actual RTC time and stores it in DATE_STRUCT.

Return Value

- None

14.4.18 _rtc_set_alarm_mqxd()

This function sets the RTC alarm.

Synopsis

```
_mqx_int _rtc_set_alarm_mqxd(DATE_STRUCT_PTR time)
```

Parameters

time [in] — The time to be set as an RTC alarm time.

Description

This function sets the RTC alarm according to DATE_STRUCT format.

Return Value

- MQX_OK

Example

The following example shows how to set the RTC alarm time to 1.1.2010, 12:30.

```
DATE_STRUCT rtc_alarm_time

rtc_alarm_time.MILLISEC = 0;
rtc_alarm_time.SECOND = 0;
rtc_alarm_time.MINUTE = 30;
rtc_alarm_time.HOUR = 12;
rtc_alarm_time.DAY = 1;
rtc_alarm_time.MONTH = 1;
rtc_alarm_time.YEAR = 2010;
_rtc_set_alarm_mqxd (&rtc_alarm_time);
```

14.4.19 _rtc_get_alarm_mqxd()

This function returns the RTC alarm time.

Synopsis

```
void _rtc_get_alarm_mqxd(DATE_STRUCT_PTR time)
```

Parameters

time [in] — The RTC alarm time.

Description

This function gets the RTC alarm time and stores it in the given DATE_STRUCT struct.

Return Value

- None

14.5 API Function Reference - IRTC Module Specific Functions

This sections serves as a function reference for the IRTC module(s).

14.5.1 _rtc_lock()

This function locks RTC registers.

Synopsis

```
void _rtc_lock(void)
```

Parameters

None

Description

This function locks RTC registers.

Return Value

- None

14.5.2 _rtc_unlock()

This function unlocks RTC registers.

Synopsis

```
void _rtc_unlock(void)
```

Parameters

none

Description

This function unlocks RTC registers.

Return Value

- None

14.5.3 _rtc_inc_upcounter()

This function increments up-counter register by 1.

Synopsis

```
void _rtc_inc_upcounter(void)
```

Parameters

none

Description

This function increments up-counter register by 1.

Return Value

- None

14.5.4 _rtc_get_upcounter()

This function returns value of the up-counter register.

Synopsis

```
uint_32 _rtc_get_upcounter(void)
```

Parameters

none

Description

This function returns value of the up-counter register.

Return Value

- The value of the up-counter register

14.5.5 _rtc_time_to_mqx_date()

This function transforms the RTC time format to the MQX date format.

Synopsis

```
void _rtc_time_to_mqx_date(
         RTC_TIME_STRUCT_PTR  rtc_time,
         DATE_STRUCT_PTR      mqx_date)
```

Parameters

rtc_time [in] — RTC time representation.

mqx_date [out] — MQX date representation.

Description

This function transforms the RTC time format to the MQX date format. Milliseconds are set to 0.

Return Value

- None

14.5.6 _rtc_time_from_mqx_date()

This function transforms the MQX date format to the RTC time format.

Synopsis

```
void _rtc_time_from_mqx_date(
         DATE_STRUCT_PTR     mqx_date,
         RTC_TIME_STRUCT_PTR  rtc_time)
```

Parameters

mqx_date [in] — MQX date representation.

rtc_time [out] — RTC time representation.

Description

This function transforms the MQX date format to the RTC time format. Milliseconds are ignored.

Return Value

- None

14.5.7 _rtc_write_to_standby_ram()

This function writes to the stand-by RAM.

Synopsis

```
_mqx_uint _rtc_write_to_standby_ram(
          uint_32           dst_address,
          uint_8            *src_ptr,
          uint_32           size)
```

Parameters

dst_address [in] — Destination address in the stand-by ram.

**src_ptr [in]* — Source data pointer.

size[in] — Number of bytes to be written.

Description

This function writes "size" in bytes pointed by "src_ptr" into the IRTC module stand-by RAM at address "dst_address".

Return Value

- MQX_OK - Operation successful
- MQX_INVALID_SIZE - Write operation failed

14.5.8 _rtc_read_from_standby_ram()

This function reads from the standby RAM.

```
_mqx_uint _rtc_read_from_standby_ram(
          uint_32            src_address,
          uint_8             *dst_ptr,
          uint_32            size)
```

Parameters

src_address [in] — Source address in the stand-by ram.

**dst_ptr [in]* — Destination data pointer.

size[in] — Number of bytes to be read.

Description

Function reads "size" in bytes from "src_address" in the stand-by RAM into "dst_ptr".

Return Value

- MQX_OK - Operation successful
- MQX_INVALID_SIZE - Read operation failed

14.5.9 _rtc_get_tamper_timestamp()

This function is specific for IRTC modules with the tamper functionality, for example MCF51EM device, and returns the last saved tamper timestamp.

Synopsis

```
void _rtc_get_tamper_timestamp(VRTC_TIME_STRUCT_PTR time)
```

Parameters

time [out] — The last saved tamper timestamp.

Description

This function returns the last saved tamper timestamp.

Return Value

- None

14.5.10 _rtc_get_tamper_status()

This function is specific to IRTC modules with the tamper functionality, for example MCF51EM device, and gets the tamper status.

Synopsis

```
RTC_TAMPER_TYPE _rtc_get_tamper_status(void)
```

Parameters

None

Description

This function returns the type of tamper detected. The value is valid when tamper interrupt status bit is set.

Return Value

- RTC_TMPR_CLEAR   00 – No tamper detected
- RTC_TMPR_PIN   01 – Tamper detected via external signal
- RTC_TMPR_BATTERY_VDDON   10 – Battery disconnected when MCU power is ON
- RTC_TMPR_BATTERY_VDDOFF   11 – Battery disconnected when MCU power is OFF

14.6 Data Types Used by the RTC Driver API

14.6.1 RTC_TIME_STRUCT

This structure is used for the RTC time interpretation and its definition can be found either in the *rtc.h* header file, for the RTC modules, or in the *irtc_mcf5xxx.h* header file for the IRTC modules).

RTC_TIME_STRUCT definition for RTC peripheral modules:

```
typedef struct rtc_time_struct
{
        uint_8 seconds;
        uint_8 minutes;
        uint_8 hours;
        uint_16 days;
}
```

RTC_TIME_STRUCT definition for IRTC peripheral modules:

```
typedef struct rtc_time_struct
{
        uint_8 seconds;
        uint_8 minutes;
        uint_8 hours;
        uint_8 days;
        uint_8 wday;
        uint_8 month;
        uint_16 year;

}
```

14.7 Example

The RTC example application that shows how to use RTC driver API functions is provided with the MQX installation and is located in the `mqx\examples\rtc` directory.

14.8 Error Codes

The RTC drivers only use the MQX I/O error codes.

Chapter 15 ESDHC Driver

15.1 Overview

This chapter describes the ESDHC device driver. The driver defines common interface for communication with various types of cards including SD, SDHC, SDIO, SDCOMBO, SDHCCOMBO, MMC, and CE-ATA. The driver is currently used as an alternative to SPI low level communication for SDCARD wrapper under the MFS stack.

15.2 Source Code Location

The source code of the ESDHC driver is located in `source\io\esdhc` directory.

15.3 Header Files

To use an ESDHC device driver, include the header files *esdhc.h* and device-specific *esdhc_xxxx.h* from `source\io\esdhc` in your application or in the BSP file *bsp.h*.

The file *esdhc_xxxx_prv.h* contains private data structures that the ESDHC device driver uses. You must include this file if you recompile an ESDHC device driver. You may also want to look at the file as you debug your application.

15.4 Installing Driver

ESDHC device driver provides an installation function ***_esdhc_install()*** that either the BSP or the application calls. The function then calls ***_io_dev_install_ext()*** internally. Installation function creates internal structures within MQX I/O subsystem and makes the driver available for public use.

ESDHC device driver installation

```
#if BSPCFG_ENABLE_ESDHC
_esdhc_install("esdhc:", &_bsp_esdhc0_init);
#endif
```

This code is located in the `/mqx/bsp/init_bsp.c` file.

15.4.1 Initialization Record

Installation function requires a pointer to initialization record to be passed to it. This record is used to initialize the device and software when the device is opened for the first time.

Synopsis

```
#include <esdhc.h>
typedef struct esdhc_init_struct
{
uint_32 CHANNEL;
uint_32 BAUD_RATE;
uint_32 CLOCK_SPEED;
} ESDHC_INIT_STRUCT, _PTR_ ESDHC_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - device number.

BAUD_RATE - desired communication baud rate.

CLOCK_SPEED - module input clock speed.

Example of ESDHC device driver initialization

```
const MCF5XXX_ESDHC_INIT_STRUCT _bsp_esdhc0_init = {
        0,                          /* ESDHC device number */
        25000000,                   /* ESDHC baudrate      */
        BSP_SYSTEM_CLOCK            /* ESDHC clock source  */
};
```

It can be found in the appropriate BSP code (*init_esdhc0.c*)

15.5 Driver Services

The table below describes the ESDHC device driver services:

API	Calls	Description
`_io_fopen()`	`_esdhc_open()`	Resets the HW module. It also applies default settings (e.g. initial 400 kHz baudrate), pin assignments, sends 80 dummy clocks, and detects the presence of the card.
`_io_fclose()`	`_esdhc_close()`	Resets the HW module.

_io_read()	_esdhc_read()	Can be called only after successful data transfer command. They return after given number of bytes was transferred. After the whole transmission, _io_fflush() should be called to wait for transfer complete flag and to check transfer errors at the host side.
_io_write()	_esdhc_write()	
_io_ioctl()	_esdhc_ioctl()	Sets up the host (card must be set up accordingly via commands over the bus). The ioctl command IO_IOCTL_ESDHC_INIT is called after _io_fopen() to determine the type of the card, to initialize it properly, and to set the baudrate requested in initialization record.

15.6 I/O Control Commands

This section describes the I/O control commands that you use when you call `_io_ioctl()`. The commands are defined in *esdhc.h*.

Command	Description
IO_IOCTL_ESDHC_INIT	Resets the HW module, sets default register values, detects the type of the card, goes through card initialization sequence, sets the baudrate according to init structure.
IO_IOCTL_ESDHC_SEND_COMMAND	Sends over the bus to card one command specified in parameter (ESDHC command structure) and returns result of the operation and card response to that command.
IO_IOCTL_ESDHC_GET_CARD	Returns type of the card detected during IO_IOCTL_ESDHC_INIT. Also detects presence of the card.
IO_IOCTL_ESDHC_GET_BAUDRATE	Returns current baudrate used.
IO_IOCTL_ESDHC_SET_BAUDRATE	Sets the baudrate given as parameter. Default baudrate is specified in initialization structure.
IO_IOCTL_ESDHC_GET_BUS_WIDTH	Returns current bus width used at the host side.

`IO_IOCTL_ESDHC_SET_BUS_WIDTH`	Sets the bus width at the host side. It should follow the successful command that sets bus width at the card. Default bus width is 1 wire.
`IO_IOCTL_FLUSH_OUTPUT`	Waits for HW transfer complete flag and checks errors at the host side. It should be called after the whole data transfer.

15.7 Send Command Structure

This section describes the ESDHC command structure used when you call `_io_ioctl()` with the `IO_IOCTL_ESDHC_SEND_COMMAND`. It is defined in *esdhc.h*.

NOTE

Not all combinations of command structure elements are valid. See SD specification or ESDHC manual for details.

```
typedef struct esdhc_command_struct
{
        uint_8  COMMAND;
        uint_32 ARGUMENT;
        uint_32 BLOCKS;
        uint_32 BLOCKSIZE;
        uint_32 RESPONSE[4];

} ESDHC_COMMAND_STRUCT, _PTR_ ESDHC_COMMAND_STRUCT_PTR;
```

Parameter	Description
COMMAND	One of the SD command definitions below.
ARGUMENT	Command-dependant argument. Argument bits must be formatted exactly according to SD specification.
BLOCKS	Number of data blocks to transfer. 0 for no data transfer commands and -1 for infinite transfers.
BLOSKZSIZE	Size of single block of the data transfer valid only if BLOCKS is not zero.
RESPONSE	Placeholder for command response from the card. For more information, please see SD specification for details.

15.7.1 Commands

This section describes the commands used in the ESDHC command structure when you call `_io_ioctl()` with the `IO_IOCTL_ESDHC_SEND_COMMAND` command. They are defined in *esdhc.h*.

Command	Description
ESDHC_CMD0	Go idle state (reset).
ESDHC_CMD1	Send operating conditions.
ESDHC_CMD2	All cards send ID.
ESDHC_CMD3	Set/send relative card ID.
ESDHC_CMD4	Set/program DSR.
ESDHC_CMD5	I/O send operating conditions.
ESDHC_CMD6	Switch check/ function.
ESDHC_CMD7	Select/deselect card.
ESDHC_CMD8	Send extended CSD.
ESDHC_CMD9	Send CSD.
ESDHC_CMD10	Send CID.
ESDHC_CMD11	Read data until stop.
ESDHC_CMD12	Stop transmission.
ESDHC_CMD13	Send card status.
ESDHC_CMD15	Go inactive state.
ESDHC_CMD16	Set block length.
ESDHC_CMD17	Read single block.
ESDHC_CMD18	Read multiple blocks.

Command	Description
ESDHC_CMD20	Write data until stop.
ESDHC_CMD24	Write block.
ESDHC_CMD25	Write multiple blocks.
ESDHC_CMD26	Program CID.
ESDHC_CMD27	Program CSD.
ESDHC_CMD28	Set write protection.
ESDHC_CMD29	Clear write protection.
ESDHC_CMD30	Send write protection.
ESDHC_CMD32	Tag sector start.
ESDHC_CMD33	Tag sector end.
ESDHC_CMD34	Untag sector.
ESDHC_CMD35	Tag erase group start.
ESDHC_CMD36	Tag erase group end.
ESDHC_CMD37	Untag erase group.
ESDHC_CMD38	Erase.
ESDHC_CMD39	Fast IO.
ESDHC_CMD40	Go IRQ state.
ESDHC_CMD42	Lock/unlock.
ESDHC_CMD52	IO R/W direct.
ESDHC_CMD53	IO R/W extended.
ESDHC_CMD55	Application specific command follows.
ESDHC_CMD56	Send/receive data block for general purpose/application specific command.
ESDHC_CMD60	R/W multiple register.
ESDHC_CMD61	R/W multiple block.
ESDHC_ACMD6	Set bus width.
ESDHC_ACMD13	Send SD status (extended).
ESDHC_ACMD22	Send number of written sectors.
ESDHC_ACMD23	Set write/erase block count.
ESDHC_ACMD41	SD application specific command send OCR.
ESDHC_ACMD42	Set/clear card detection.
ESDHC_ACMD51	Send SCR.

15.8 Card Types

This section describes the card types which are returned as a parameter when you call `_io_ioctl()` with the `IO_IOCTL_ESDHC_GET_CARD` command. They are defined in *esdhc.h*.

Flag	Description
ESDHC_CARD_NONE	No card detected in the slot.
ESDHC_CARD_UNKNOWN	Card not initialized yet or not recognized.
ESDHC_CARD_SD	SD normal capacity memory card detected in the slot.
ESDHC_CARD_SDHC	SD high capacity memory card detected in the slot.
ESDHC_CARD_SDIO	SDIO card detected in the slot.
ESDHC_CARD_SDCOMBO	SDIO card with SD normal capacity memory capability detected in the slot.
ESDHC_CARD_SDHCCOMBO	SDIO card with SD high capacity memory capability detected in the slot.
ESDHC_CARD_MMC	MMC card detected in the slot.
ESDHC_CARD_CEATA	CE-ATA card detected in the slot.

15.9 Bus Widths

This section describes the bus widths that you use when you call `_io_ioctl()` with the `IO_IOCTL_ESDHC_SET_BUS_WIDTH` command. They are defined in *esdhc.h*.

Flag	Description
ESDHC_BUS_WIDTH_1BIT	1-wire data transfer (supported by all cards).
ESDHC_BUS_WIDTH_4BIT	4-wire data transfer (optional for SDIO cards).
ESDHC_BUS_WIDTH_8BIT	8-wire data transfer (MMC cards only).

15.10 Error Codes

The ESDHC device driver defines the following error codes.

Error code	Description
ESDHC_OK	Success.
ESDHC_ERROR_INIT_FAILED	Error during card initialization.
ESDHC_ERROR_COMMAND_FAILED	Error during command execution over the bus.

ESDHC_ERROR_COMMAND_TIMEOUT	No response from the card to the command.
ESDHC_ERROR_DATA_TRANSFER	Error during data transfer detected at the host side and returned by IO_IOCTL_FLUSH_OUTPUT.
ESDHC_ERROR_INVALID_BUS_WIDTH	Wrong bus width detected during get/set at the host side.

15.11 Example

```
FILE_PTR esdhc_fd;
ESDHC_COMMAND_STRUCT command;
boolean sdhc;
uint_32 param, rca, sector;
uint_8 buffer[512];

/* Open ESDHC driver */
esdhc_fd = fopen ("esdhc:", NULL);
if (NULL == esdhc_fd)
{
        _task_block ();
}

/* Initialize and detect card */
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_INIT, NULL))
{
        _task_block ();
}



/* SDHC check */
sdhc = FALSE;
param = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_GET_CARD, &param))
{
        _task_block ();
}
if ((ESDHC_CARD_SD == param) || (ESDHC_CARD_SDHC == param) || (ESDHC_CARD_SDCOMBO ==
param) || (ESDHC_CARD_SDHCCOMBO == param))
{
        if ((ESDHC_CARD_SDHC == param) || (ESDHC_CARD_SDHCCOMBO == param))
        {
                sdhc = TRUE;
        }
}
else
{
        /* Not SD memory card */
        _task_block ();
}
```

```
/* Card identify */
command.COMMAND = ESDHC_CMD2;
command.ARGUMENT = 0;
command.BLOCKS = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
        _task_block ();
}

/* Get card relative address */
command.COMMAND = ESDHC_CMD3;
command.ARGUMENT = 0;
command.BLOCKS = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
        _task_block ();
}
rca = command.RESPONSE[0] & 0xFFFF0000;

/* Select card */
command.COMMAND = ESDHC_CMD7;
command.ARGUMENT = rca;
command.BLOCKS = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
        _task_block ();
}

/* Application specific command */
command.COMMAND = ESDHC_CMD55;
command.ARGUMENT = rca;
command.BLOCKS = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
        _task_block ();
}

/* Set bus width 4 */
command.COMMAND = ESDHC_ACMD6;
command.ARGUMENT = 2;
command.BLOCKS = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
        _task_block ();
}

param = ESDHC_BUS_WIDTH_4BIT;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SET_BUS_WIDTH, &param))
{
        _task_block ();
}
```

```
/* SD card data address adjustment */
sector = 0;
if (! sdhc)
{
        sector <<= 9;
}

/* Read block command */
command.COMMAND = ESDHC_CMD17;
command.ARGUMENT = sector;
command.BLOCKS = 1;
command.BLOCKSIZE = 512;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
        _task_block ();
}

/* Read sector 0 */
if (512 != fread (buffer, 1, 512, esdhc_fd))
{
        _task_block ();
}

/* Wait for transfer complete and check errors at host side */
if (ESDHC_OK != fflush (esdhc_fd))
{
        _task_block ();
}

/* Close driver */
fclose (esdhc_fd);
```

Chapter 16 FlexCAN Driver

16.1 Overview

This section describes the FlexCAN driver that accompanies the MQX release. Unlike other drivers in the MQX release, FlexCAN driver implements custom C-language API instead of standard MQX I/O Subsystem (POSIX) driver interface.

16.2 Source Code Location

The source files for the FlexCAN driver are located in `source\io\can\flexcan` directory. It contains generic files and device-specific source files that are named according to the platform supported.

16.3 Header Files

To use the FlexCAN driver, include the header file named *flexcan.h* into your application.

16.4 API Function Reference - FlexCAN Module Related Functions

This section provides function reference for the FlexCAN module driver.

NOTE

The general term "mailbox" corresponds to Message Buffer in FlexCAN Reference Manual terminology.

16.4.1 FLEXCAN_Softreset()

This function (re)initializes the FlexCAN module.

Synopsis

```
uint_32 FLEXCAN_Softreset(
                uint_8 dev_num)
```

Parameters

dev_num [in] — FlexCAN device number

Description

The function performs software reset of the FlexCAN module and disables/halts it as a preparation for the subsequent module setup.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_SOFTRESET_FAILED (reset failed)

Example

```
/* reset FlexCAN module 0 */
uint_32 result = FLEXCAN_Softreset(0);
```

16.4.2 FLEXCAN_Start()

This function puts the FlexCAN module into a working state.

Synopsis

```
uint_32 FLEXCAN_Start(
                uint_8 dev_num)
```

Parameters

dev_num [in] — FlexCAN device number

Description

The function enables the FlexCAN module. It is called after the module is set up.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* start FlexCAN module 0 */
uint_32 result = FLEXCAN_Start(0);
```

16.4.3 FLEXCAN_Get_msg_object()

This function returns the pointer to the specified message buffer register memory area.

Synopsis

```
pointer FLEXCAN_Get_msg_object(
                uint_8  dev_num,
                uint_32 mailbox_number)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

Description

The function returns the pointer to the base address of the specified message buffer within the register memory area. The mailbox can be directly accessed using the structure `FLEXCAN_MSG_OBJECT_STRUCT`.

Return Value

- valid address (success)
- NULL (error)

Example

```
/* get mailbox 15 address */
FLEXCAN_MSG_OBJECT_STRUCT mailbox = FLEXCAN_Get_msg_object(0,15);
```

16.4.4 FLEXCAN_Select_mode()

This function selects the mode of operation of the FlexCAN module.

Synopsis

```
uint_32 FLEXCAN_Select_mode(
                uint_8  dev_num,
                uint_32 mode)
```

Parameters

dev_num [in] – FlexCAN device number

mode [in] – FlexCAN mode of operation

Description

The function selects the mode of operation of the FlexCAN module. Available modes are:

- FLEXCAN_NORMAL_MODE (starts normal operation)
- FLEXCAN_LISTEN_MODE (puts device into listen only mode)
- FLEXCAN_TIMESYNC_MODE (free running timer synchronization mode)
- FLEXCAN_LOOPBK_MODE (loopback mode)
- FLEXCAN_BOFFREC_MODE (automatic recovery from the bus off state)
- FLEXCAN_FREEZE_MODE (halt/freeze mode for debugging)
- FLEXCAN_DISABLE_MODE (FlexCAN disabled)

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MODE (wrong operating mode)

Example

```
/* select normal mode for FlexCAN module 0 */
uint_32 result = FLEXCAN_Select_mode(0,FLEXCAN_NORMAL_MODE);
```

16.4.5 FLEXCAN_Select_clk()

This function selects the input clock source for the FlexCAN module.

Synopsis

```
uint_32 FLEXCAN_Select_clk(
                uint_8  dev_num,
                uint_32 clk)
```

Parameters

dev_num [in] - FlexCAN device number

clk [in] - FlexCAN clock source

Description

The function selects the input clock source for the FlexCAN module. Available clock sources are:

- FLEXCAN_IPBUS_CLK (internal bus clock)
- FLEXCAN_OSC_CLK (EXTAL clock source)

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_CLOCK_SOURCE_INVALID (wrong clock source)

Example

```
/* set FlexCAN clock source to internal bus */
uint_32 result = FLEXCAN_Select_clk(0,FLEXCAN_IPBUS_CLK);
```

16.4.6 FLEXCAN_Initialize()

This is the main setup function of the FlexCAN module.

Synopsis

```
uint_32 FLEXCAN_Initialize(
                uint_8  dev_num,
                uint_32 bit_timing0,
                uint_32 bit_timing1,
                uint_32 frequency,
                uint_32 clk)
```

Parameters

dev_num [in] - FlexCAN device number

bit_timing0 [in] - FlexCAN PSEG1 and PROPSEG settings

bit_timing1 [in] - FlexCAN PSEG2, RJW, and PRESDIV settings

frequency [in] - Desired bus baudrate in kb/s

clk [in] - FlexCAN clock source (see function FLEXCAN_Select_mode())

Description

The function performs the software reset of the FlexCAN module, disables it, sets up the clock sources and bit timings, clears all acceptance masks, and resets all mailboxes. The hardware remains in the disabled mode after the function returns.

There are two ways of using this function:

1. Parameters *bit_timing0* and *bit_timing1* set to 0 - this instructs the function to use predefined bit timing settings according to given frequency and clock source. There are available predefined settings for all currently supported boards.
2. Parameters *bit_timing0* and *bit_timing1* are non zero - the function will set up bit timing according these settings which must be coded in the following way:
   bit_timing0 = (PSEG1 << 16) | PROPSEG;
   bit_timing1 = (PSEG2 << 16) | (RJW << 8) | PRESDIV;
   The values are directly written to the CANCTRL register without any change.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INIT_FAILED (module reset failed)
- FLEXCAN_INVALID_FREQUENCY (wrong clock source)

Example

```
/* initialize FlexCAN module 0 to 250 kbit/s and internal bus clock source */
uint_32 result = FLEXCAN_Initialize(0,0,0,250,FLEXCAN_IPBUS_CLK);
```

16.4.7 FLEXCAN_Initialize_mailbox()

This function sets up one FlexCAN message buffer.

Synopsis

```
uint_32 FLEXCAN_Initialize_mailbox(
                uint_8  dev_num,
                uint_32 mailbox_number,
                uint_32 identifier,
                uint_32 data_len_code,
                uint_32 direction,
                uint_32 format,
                uint_32 int_enable)
```

Parameters

dev_num [in] - FlexCAN device number

mailbox_number [in] - FlexCAN message buffer index

identifier[in] - FlexCAN message identifier bits

data_len_code [in] - Number of bytes transferred (0-8)

direction [in] - Transmits or receives (FLEXCAN_TX or FLEXCAN_RX)

format [in] - FlexCAN message format (FLEXCAN_STANDARD or FLEXCAN_EXTENDED)

int_enable [in] - Whether to enable interrupt for message buffer (FLEXCAN_ENABLE or FLEXCAN_DISABLE)

Description

The function (re)initializes particular FlexCAN message buffer using the given information. Message buffer remains inactive after the function returns.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)
- FLEXCAN_DATA_SIZE_ERROR (wrong data length)
- FLEXCAN_INVALID_DIRECTION (wrong transmission direction)
- FLEXCAN_MESSAGE_FORMAT_UNKNOWN (wrong message format)
- FLEXCAN_INT_ENABLE_FAILED (interrupt enable failed)
- FLEXCAN_INT_DISABLE_FAILED (interrupt disable failed)

Example

```
/* setup mailbox 15 to transmit standard ID 0x7FF, 8 byte data and enable particular
interrupt */
uint_32 result = FLEXCAN_Initialize_mailbox
(0,15,0x7FF,8,FLEXCAN_TX,FLEXCAN_STANDARD,FLEXCAN_ENABLE);
```

16.4.8 FLEXCAN_Request_mailbox()

This function sets up one FlexCAN message buffer to be used as remote frame initiated by the FlexCAN module.

Synopsis

```
uint_32 FLEXCAN_Request_mailbox(
                uint_8  dev_num,
                uint_32 mailbox_number,
                uint_32 format)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

format [in] – FlexCAN message format (FLEXCAN_STANDARD or FLEXCAN_EXTENDED)

Descriptio

The function sets the RTR bit for particular FlexCAN message buffer.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)

Example

```
/* turn previously set FlexCAN mailbox 15 for remote frame requesting */
uint_32 result = FLEXCAN_Request_mailbox(0,15,FLEXCAN_STANDARD);
```

16.4.9 FLEXCAN_Activate_mailbox()

This function activates one FlexCAN message buffer so it participates on the bus arbitration.

Synopsis

```
uint_32 FLEXCAN_Activate_mailbox(
                uint_8  dev_num,
                uint_32 mailbox_number,
                uint_32 code_val)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

code_val [in] – FlexCAN message buffer codes/status bits

Description

The function sets the FlexCAN message buffer code/status bits.

Available codes for TX buffers:

- FLEXCAN_TX_MSG_BUFFER_NOT_ACTIVE (does not participate on the bus)
- FLEXCAN_MESSAGE_TRANSMIT_ONCE (data frame sent once)
- FLEXCAN_MESSAGE_TRANSMIT_REMOTE (remote frame sent once)
- FLEXCAN_MESSAGE_TRANSMIT_RESPONED (transmit response to remote frame)
- FLEXCAN_MESSAGE_TRANSMIT_RESPONED_ONLY (transmit response now)

Available codes for RX buffers:

- FLEXCAN_RX_MSG_BUFFER_NOT_ACTIVE (does not participate on the bus)
- FLEXCAN_RX_MSG_BUFFER_EMPTY (active and waiting)
- FLEXCAN_RX_MSG_BUFFER_FULL (active and received data)
- FLEXCAN_RX_MSG_BUFFER_OVERRUN (received again, not read)
- FLEXCAN_RX_MSG_BUFFER_BUSY (data are filled in right now)

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)

Example

```
/* activate previously set FlexCAN mailbox 15 to send message once */
uint_32 result = FLEXCAN_Activate_mailbox(0,15,FLEXCAN_MESSAGE_TRANSMIT_ONCE);
```

16.4.10 FLEXCAN_Lock_mailbox()

This function locks one FlexCAN message buffer so it can be accessed by the system.

Synopsis

```
uint_32 FLEXCAN_Lock_mailbox(
               uint_8  dev_num,
               uint_32 mailbox_number)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

Description

The function locks the FlexCAN message buffer. It must be used before any mailbox access.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)

Example

```
/* lock FlexCAN mailbox 15 */
uint_32 result = FLEXCAN_Lock_mailbox(0,15);
```

16.4.11 FLEXCAN_Unlock_mailbox()

This function unlocks all FlexCAN message buffers.

Synopsis

```
uint_32 FLEXCAN_Unlock_mailbox(
                uint_8  dev_num)
```

Parameters

dev_num [in] – FlexCAN device number

Description

The function unlocks all FlexCAN message buffers.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* unlock all FlexCAN mailboxes */
uint_32 result = FLEXCAN_Unlock_mailbox(0);
```

16.4.12 FLEXCAN_Set_global_extmask()

This function sets global extended ID filtering mask for FlexCAN message buffers 0-13.

Synopsis

```
uint_32 FLEXCAN_Set_global_extmask(
                uint_8  dev_num,
                uint_32 extmask)
```

Parameters

dev_num [in] – FlexCAN device number

extmask [in] – Extended ID bit mask

Description

The function sets the global extended ID filtering mask for active FlexCAN message buffers 0-13. The '1' bit within the extmask specifies the bit-positions in the extended ID of messages on the bus that must match the corresponding extended ID bits of the active FlexCAN message buffers in order to receive the message. The '0' bit means don't care.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* set global extended mask to don't care about least significant ID bit */
uint_32 result = FLEXCAN_Set_global_extmask(0,0x1FFFFFFE);
```

16.4.13 FLEXCAN_Set_buf14_extmask()

This function sets the extended ID filtering mask for FlexCAN message buffer 14.

Synopsis

```
uint_32 FLEXCAN_Set_buf14_extmask(
                uint_8  dev_num,
                uint_32 extmask)
```

Parameters

dev_num [in] – FlexCAN device number

extmask [in] – Extended ID bit mask

Description

The function sets the extended ID filtering mask for active FlexCAN message buffer 14.

- 1 bit within the extmask – Specifies the bit-positions in the extended ID of messages on the bus that must match the corresponding extended ID bits of the active FlexCAN message buffer 14 in order to receive the message.
- 0 bit – It is a don't care bit.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* set mailbox 14 extended mask to don't care about least significant ID bit */
uint_32 result = FLEXCAN_Set_buf14_extmask(0,0x1FFFFFFE);
```

16.4.14 FLEXCAN_Set_buf15_extmask()

This function sets the extended ID filtering mask for FlexCAN message buffer 15.

Synopsis

```
uint_32 FLEXCAN_Set_buf15_extmask(
                uint_8  dev_num,
                uint_32 extmask)
```

Parameters

dev_num [in] – FlexCAN device number

extmask [in] – Extended ID bit mask

Description

The function sets the extended ID filtering mask for FlexCAN message buffer 15.

1 bit within the extmask – Specifies the bit-positions in the extended ID of messages on the bus that must match the corresponding extended ID bits of the active FlexCAN message buffer 15 to receive the message.

0 bit – It is a don't care bit.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* set mailbox 15 extended mask to don't care about least significant ID bit */
uint_32 result = FLEXCAN_Set_buf15_extmask(0,0x1FFFFFFE);
```

16.4.15 FLEXCAN_Set_global_stdmask()

This function sets the global standard ID filtering mask for FlexCAN message buffers 0-13.

Synopsis

```
uint_32 FLEXCAN_Set_global_stdmask(
                uint_8  dev_num,
                uint_32 stdmask)
```

Parameters

dev_num [in] – FlexCAN device number

stdmask [in] – Standard ID bit mask

Description

The function sets the global standard ID filtering mask for all active FlexCAN message buffers 0-13.

1 bit within the stdmask – Specifies the bit-positions in the standard ID of messages on the bus that must match the corresponding standard ID bits of the active FlexCAN message buffers in order to receive the message.

0 bit – It is a don't care bit.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* set global standard mask to don't care about least significant ID bit */
uint_32 result = FLEXCAN_Set_global_stdtmask(0,0x7FE);
```

16.4.16 FLEXCAN_Set_buf14_stdmask()

This function sets the standard ID filtering mask for FlexCAN message buffer 14.

Synopsis

```
uint_32 FLEXCAN_Set_buf14_stdmask(
                uint_8  dev_num,
                uint_32 stdmask)
```

Parameters

dev_num [in] – FlexCAN device number.

stdmask [in] – Standard ID bit mask.

Description

The function sets standard ID filtering mask for active FlexCAN message buffer 14.

1 bit within the stdmask – Specifies the bit-positions in the standard ID of messages on the bus that must match the corresponding standard ID bits of the active FlexCAN message buffer 14 in order to receive the message.

0 bit – It is a don't care bit.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* set mailbox 14 standard mask to don't care about least significant ID bit */
uint_32 result = FLEXCAN_Set_buf14_stdmask(0,0x7FE);
```

16.4.17 FLEXCAN_Set_buf15_stdmask()

This function sets the standard ID filtering mask for FlexCAN message buffer 15.

Synopsis

```
uint_32 FLEXCAN_Set_buf15_stdmask(
                uint_8  dev_num,
                uint_32 stdmask)
```

Parameters

dev_num [in] – FlexCAN device number

stdmask [in] – Standard ID bit mask

Description

The function sets the standard ID filtering mask for active FlexCAN message buffer 15.

1 bit – Specifies the bit-positions in the standard ID of messages on the bus that must match the corresponding standard ID bits of the active FlexCAN message buffer 15 in order to receive the message.
0 bit – It is a don't care bit.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* set mailbox 15 standard mask to don't care about least significant ID bit */
uint_32 result = FLEXCAN_Set_buf15_stdmask(0,0x7FE);
```

16.4.18 FLEXCAN_Tx_successful()

This function checks whether any message was transmitted.

Synopsis

```
boolean FLEXCAN_Tx_successful(
                uint_8  dev_num)
```

Parameters

dev_num [in] – FlexCAN device number

Description

The function returns TRUE if any message buffer interrupt flag is set.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* get TX successful flag */
boolean result = FLEXCAN_Tx_successful(0);
```

16.4.19 FLEXCAN_Tx_mailbox()

This function transmits given data using the already set up FlexCAN mailbox.

Synopsis

```
uint_32 FLEXCAN_Tx_mailbox(
                uint_8  dev_num,
                uint_32 mailbox_number,
                pointer data)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

data [in] – Pointer to input data buffer

Description

The function transmits message once. The mailbox must already be set up. The length of the input data buffer must correspond to the mailbox data length.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)

Example

```
/* send data using message buffer 15 */
uint_32 result = FLEXCAN_Tx_mailbox(0,15,data_ptr);
```

16.4.20 FLEXCAN_Rx_mailbox()

This function gets data from the given FlexCAN mailbox.

Synopsis

```
uint_32 FLEXCAN_Rx_mailbox(
                uint_8  dev_num,
                uint_32 mailbox_number,
                pointer data)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

data [out] – Pointer to output data buffer

Description

The function receives data from a given message buffer. User should check the error codes for appropriate handling. The mailbox is again activated and prepared for further receiving.

Return Value

- FLEXCAN_OK (data received, success)
- FLEXCAN_MESSAGE_BUSY (data received, but the state was busy)
- FLEXCAN_MESSAGE_LOST (data received, but one or more messages were lost)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)
- FLEXCAN_NO_MESSAGE (mailbox is empty)

Example

```
/* receive data from message buffer 15 */
uint_32 result = FLEXCAN_Rx_mailbox(0,15,data_ptr);
```

16.4.21 FLEXCAN_Disable_mailbox()

This function removes the given FlexCAN mailbox from participating on the bus arbitration.

Synopsis

```
uint_32 FLEXCAN_Disable_mailbox(
                uint_8  dev_num,
                uint_32 mailbox_number)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

Description

The function disables the given mailbox so it no longer participates in bus arbitration.

Return Value

- FLEXCAN_OK (data received, success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)

Example

```
/* disable message buffer 15 */
uint_32 result = FLEXCAN_Disable_mailbox(0,15);
```

16.4.22 FLEXCAN_Request_message()

This function sets up and activates one FlexCAN message buffer to be used as a remote frame initiated by the FlexCAN module.

Synopsis

```
uint_32 FLEXCAN_Request_message(
                uint_8  dev_num,
                uint_32 mailbox_number,
                uint_32 format)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

format [in] – FlexCAN message format (FLEXCAN_STANDARD or FLEXCAN_EXTENDED)

Description

The function calls FLEXCAN_Request_mailbox() and then activates the mailbox accordingly so the remote frame is sent. The mailbox parameters have to be set up prior to calling this function.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)

Example

```
/* send remote frame request using previously initialized FlexCAN mailbox 15 */
uint_32 result = FLEXCAN_Request_message(0,15,FLEXCAN_STANDARD);
```

16.4.23 FLEXCAN_Rx_message()

This function gets data and other information from the given FlexCAN Rx mailbox.

Synopsis

```
uint_32 FLEXCAN_Rx_message(
                uint_8      dev_num,
                uint_32     mailbox_number,
                uint_32_ptr identifier,
                uint_32     format,
                uint_32_ptr data_len_code,
                pointer     data,
                uint_32     int_enabled)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

identifier [out] – ID from the message buffer

format [in] – Message buffer ID format (FLEXCAN_STANDARD or FLEXCAN_EXTENDED)

data_len_code [out] – Received data length

data [out] – Received data

int_enabled [int] – Used to unlock mailbox in non-interrupt mode (FLEXCAN_ENABLE or FLEXCAN_DISABLE)

Description

The function returns data, data length, and ID of the received message from given mailbox. Always check the error codes for appropriate handling. The mailbox is again activated and prepared for further receiving.

Return Value

- FLEXCAN_OK (data received, success)
- FLEXCAN_MESSAGE_OVERWRITTEN (data received, but one or more messages were lost)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)
- FLEXCAN_NO_MESSAGE (mailbox is empty)
- FLEXCAN_MESSAGE_FORMAT_UNKNOWN (wrong message format)

Example

```
/* receive data, length and ID from message buffer 15 and unlock it */
uint_32 result = FLEXCAN_Rx_message(0,15,&id,FLEXCAN_STANDARD,&len,
data_ptr,FLEXCAN_DISABLE);
```

16.4.24 FLEXCAN_Tx_message()

This function sends the specified message using the given FlexCAN transmit mailbox.

Synopsis

```
uint_32 FLEXCAN_Tx_message(
                uint_8  dev_num,
                uint_32 mailbox_number,
                uint_32 identifier,
                uint_32 format,
                uint_32 data_len_code,
                pointer data)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

identifier [in] – Message buffer ID to use

format [in] – Message buffer ID format (FLEXCAN_STANDARD or FLEXCAN_EXTENDED)

data_len_code [in] – Data length

data [in] – Transmitted data buffer

Description

The function either sends the message once, or it responds to a remote frame by using the given mailbox number and specified parameters. Mailbox must be set up prior to calling this function.

Return Value

- FLEXCAN_OK (data received, success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)
- FLEXCAN_DATA_SIZE_ERROR (data length not in range 0..8 bytes)
- FLEXCAN_MESSAGE_FORMAT_UNKNOWN (wrong message format)

Example

```
/* transmit message once using mailbox 15 */
uint_32 result = FLEXCAN_Tx_message(0,15,id,FLEXCAN_STANDARD,8,data_ptr);
```

16.4.25 FLEXCAN_Read()

This function reads 32-bit value from within the FlexCAN module register space.

Synopsis

```
uint_32 FLEXCAN_Read(
                uint_8      dev_num,
                uint_32     offset,
                uint_32_ptr data_ptr)
```

Parameters

dev_num [in] – FlexCAN device number

offset [in] – FlexCAN register offset

data_ptr [out] – Where to store the result

Description

The function reads 32-bit value from the FlexCAN module register space specified by an offset to a device register base.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* Read ID of the first message buffer register */
uint_32 result = FLEXCAN_Read(0,FLEXCAN_MSG_BUFADDR_OFFSET+4,data_ptr);
```

16.4.26 FLEXCAN_Write()

This function writes 32-bit value to the specified FlexCAN module register space.

Synopsis

```
uint_32 FLEXCAN_Write(
                uint_8  dev_num,
                uint_32 offset,
                uint_32 value)
```

Parameters

dev_num [in] – FlexCAN device number

offset [in] – FlexCAN register offset

value [in] – 32 bit value to be written

Description

This function writes 32-bit value to the FlexCAN module register space specified by an offset to a device register base.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* Write ID of the first message buffer register */
uint_32 result = FLEXCAN_Write(0,FLEXCAN_MSG_BUFADDR_OFFSET+4,0);
```

16.4.27 FLEXCAN_Get_status()

This function reads the 32-bit value from the FlexCAN module register ERRSTAT.

Synopsis

```
uint_32 FLEXCAN_Get_status(
                uint_8      dev_num,
                uint_32_ptr can_status)
```

Parameters

dev_num [in] – FlexCAN device number

can_status [out] – Where to store the result

Description

The function reads 32-bit status value from the FlexCAN module register ERRSTAT.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)

Example

```
/* Read status */
uint_32 result = FLEXCAN_Get_status(0,data_ptr);
```

16.4.28 FLEXCAN_Update_message()

This function updates the FlexCAN mailbox used as a remote response.

Synopsis

```
uint_32 FLEXCAN_Update_message(
                uint_8  dev_num,
                pointer data_ptr,
                uint_32 data_len_code,
                uint_32 format,
                uint_32 mailbox_number)
```

Parameters

dev_num [in] – FlexCAN device number

data_ptr [in] – Response data

data_len_code [in] – Response data length

format [in] – Message buffer ID format (FLEXCAN_STANDARD or FLEXCAN_EXTENDED)

mailbox_number[in] – FlexCAN message buffer index

Description

The function updates the data in the message buffer previously set up as a response to the remote frames over the bus.

Return Value

- FLEXCAN_OK (data received, success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)
- FLEXCAN_DATA_SIZE_ERROR (data length not in range 0..8 bytes)
- FLEXCAN_RTR_NOT_SET (mailbox not set as remote response)

Example

```
/* update data in mailbox 15 used as remote response */
uint_32 result = FLEXCAN_Update_message(0,data_ptr,8,FLEXCAN_STANDARD,15);
```

16.4.29 FLEXCAN_Int_enable()

This function initializes and enables the interrupt for the specified FlexCAN mailbox.

Synopsis

```
uint_32 FLEXCAN_Int_enable(
                uint_8  dev_num,
                uint_32 mailbox_number)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

Description

The function initializes the FlexCAN message buffer interrupt in MQX and enables the specified message buffer interrupt source.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)
- FLEXCAN_INT_ENABLE_FAILED (wrong interrupt vector)

Example

```
/* enable interrupt for message buffer 15 */
uint_32 result = FLEXCAN_Int_enable(0,15);
```

16.4.30 FLEXCAN_Error_int_enable()

This function initializes and enables the FlexCAN error interrupt.

Synopsis

```
uint_32 FLEXCAN_Error_int_enable(
                uint_8  dev_num)
```

Parameters

dev_num [in] – FlexCAN device number

Description

The function initializes the FlexCAN error interrupt in MQX and enables the specified interrupt source.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INT_ENABLE_FAILED (wrong interrupt vector)

Example

```
/* enable error interrupt */
uint_32 result = FLEXCAN_Error_int_enable(0);
```

16.4.31 FLEXCAN_Int_disable()

This function disables the interrupt for the specified FlexCAN mailbox.

Synopsis

```
uint_32 FLEXCAN_Int_disable(
               uint_8  dev_num,
               uint_32 mailbox_number)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

Description

The function de-initializes the FlexCAN message buffer interrupt in MQX and disables the specified message buffer interrupt source.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)
- FLEXCAN_INT_DISABLE_FAILED (wrong interrupt vector)

Example

```
/* disable interrupt for message buffer 15 */
uint_32 result = FLEXCAN_Int_disable(0,15);
```

16.4.32 FLEXCAN_Error_int_disable()

This function disables the FlexCAN error interrupt.

Synopsis

```
uint_32 FLEXCAN_Error_int_disable(
                uint_8  dev_num)
```

Parameters

dev_num [in] – FlexCAN device number

Description

The function de-initializes the FlexCAN error interrupt in MQX and disables the specified interrupt source.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INT_DISABLE_FAILED (wrong interrupt vector)

Example

```
/* disable error interrupt */
uint_32 result = FLEXCAN_Error_int_disable(0);
```

16.4.33 FLEXCAN_Install_isr()

This function installs the interrupt service routine for the specified FlexCAN mailbox.

Synopsis

```
uint_32 FLEXCAN_Install_isr(
                uint_8  dev_num,
                uint_32 mailbox_number,
                pointer isr)
```

Parameters

dev_num [in] – FlexCAN device number

mailbox_number [in] – FlexCAN message buffer index

isr [in] – Interrupt service routine address

Description

The function installs the interrupt service routine within MQX for FlexCAN message buffer TX or RX requests.

NOTE

On some systems all message buffers share the same interrupt vector. Therefore, this function installs one routine for all message buffers at once.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)
- FLEXCAN_INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void my_isr_function (pointer can_reg_base_ptr);

/* install interrupt service routine for message buffer 15 */
uint_32 result = FLEXCAN_Install_isr(0,15,my_isr_function);
```

16.4.34 FLEXCAN_Install_isr_err_int()

This function installs the FlexCAN error interrupt service routine.

Synopsis

```
uint_32 FLEXCAN_Install_isr_err_int(
                uint_8  dev_num,
                pointer isr)
```

Parameters

dev_num [in] – FlexCAN device number

isr [in] – Interrupt service routine address

Description

The function installs the FlexCAN error interrupt service routine within MQX.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void my_err_isr_function (pointer can_reg_base_ptr);

/* install error interrupt service routine */
uint_32 result = FLEXCAN_Install_isr_err_int(0,my_err_isr_function);
```

16.4.35 FLEXCAN_Install_isr_boff_int()

This function installs the FlexCAN bus off interrupt service routine.

Synopsis

```
uint_32 FLEXCAN_Install_isr_boff_int(
                uint_8  dev_num,
                pointer isr)
```

Parameters

dev_num [in] – FlexCAN device number.

isr [in] – Interrupt service routine address.

Description

The function installs the FlexCAN bus off interrupt service routine within MQX.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void my_boff_isr_function (pointer can_reg_base_ptr);

/* install bus off interrupt service routine */
uint_32 result = FLEXCAN_Install_isr_boff_int(0,my_boff_isr_function);
```

16.4.36 FLEXCAN_Install_isr_wake_int()

This function installs the FlexCAN wake up interrupt service routine.

Synopsis

```
uint_32 FLEXCAN_Install_isr_wake_int(
                uint_8  dev_num,
                pointer isr)
```

Parameters

dev_num [in] – FlexCAN device number

isr [in] – Interrupt service routine address

Description

The function installs the FlexCAN wake up interrupt service routine within MQX where available.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void my_wake_isr_function (pointer can_reg_base_ptr);

/* install wake up interrupt service routine */
uint_32 result = FLEXCAN_Install_isr_wake_int(0,my_wake_isr_function);
```

16.4.37 FLEXCAN_Int_status()

This function returns the FlexCAN interrupt status.

Synopsis

```
uint_32 FLEXCAN_Int_status(
                uint_8  dev_num)
```

Parameters

dev_num [in] – FlexCAN device number

Description

The function returns the interrupt status of the specified FlexCAN module based on the value of ERRSTAT register.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_TX_RX_INT (any message buffer interrupt pending)
- FLEXCAN_ERROR_INT (error interrupt pending)
- FLEXCAN_BUSOFF_INT (bus off interrupt pending)
- FLEXCAN_WAKEUP_INT (wake up interrupt pending)

Example

```
/* get interrupt status */
uint_32 result = FLEXCAN_Int_status(0);
```

16.5 Data Types

This section describes the data types used by the FlexCAN driver API.

16.5.1 FLEXCAN_MSG_OBJECT_STRUCT

This structure can be used to access the FlexCAN message buffer register space directly.

```
typedef struct mcfxxxx_flexcan_msg_struct
{
        uint_32 CONTROL;
        uint_32 ID;
        uchar   DATA[8];
} MCFXXXX_FCAN_MSG_STRUCT, _PTR_ MCFXXXX_FCAN_MSG_STRUCT_PTR;
typedef volatile struct mcfxxxx_flexcan_msg_struct VMCFXXXX_FCAN_MSG_STRUCT;
typedef volatile struct mcfxxxx_flexcan_msg_struct _PTR_
VMCFXXXX_FCAN_MSG_STRUCT_PTR;
typedef VMCFXXXX_FCAN_MSG_STRUCT      FLEXCAN_MSG_OBJECT_STRUCT;
typedef VMCFXXXX_FCAN_MSG_STRUCT_PTR  FLEXCAN_MSG_OBJECT_STRUCT_PTR;
```

16.6 Error Codes

The FlexCAN driver defines the following error codes:

Error code	Description
FLEXCAN_OK	Success
FLEXCAN_UNDEF_ERROR	Unknown error
FLEXCAN_MESSAGE14_TX	Wrong mailbox 14 usage
FLEXCAN_MESSAGE15_TX	Wrong mailbox 15 usage
FLEXCAN_MESSAGE_OVERWRITTEN	Previously received message lost
FLEXCAN_NO_MESSAGE	No message received
FLEXCAN_MESSAGE_LOST	Previously received message lost
FLEXCAN_MESSAGE_BUSY	Message buffer updated at the moment
FLEXCAN_MESSAGE_ID_MISSMATCH	Wrong ID detected
FLEXCAN_MESSAGE14_START	Wrong mailbox 14 usage
FLEXCAN_MESSAGE15_START	Wrong mailbox 15 usage
FLEXCAN_INVALID_ADDRESS	Wrong device specified
FLEXCAN_INVALID_MAILBOX	Wrong message buffer index
FLEXCAN_TIMEOUT	Time-out occurred
FLEXCAN_INVALID_FREQUENCY	Wrong frequency setting

Error code	Description
FLEXCAN_INT_ENABLE_FAILED	MQX interrupt enabling failed
FLEXCAN_INT_DISABLE_FAILED	MQX interrupt disabling failed
FLEXCAN_INT_INSTALL_FAILED	MQX interrupt installation failed
FLEXCAN_REQ_MAILBOX_FAILED	Error requesting message
FLEXCAN_DATA_SIZE_ERROR	Data length not in range 0..8
FLEXCAN_MESSAGE_FORMAT_UNKNOWN	Wrong message format specified
FLEXCAN_INVALID_DIRECTION	TX via RX buffer or vice versa
FLEXCAN_RTR_NOT_SET	Message buffer not set as remote request
FLEXCAN_SOFTRESET_FAILED	Software reset failed
FLEXCAN_INVALID_MODE	Wrong operating mode specified
FLEXCAN_START_FAILED	Error during FlexCAN start
FLEXCAN_CLOCK_SOURCE_INVALID	Wrong clock source specified
FLEXCAN_INIT_FAILED	Error during FlexCAN reset
FLEXCAN_ERROR_INT_ENABLE_FAILED	MQX interrupt enabling failed
FLEXCAN_ERROR_INT_DISABLE_FAILED	MQX interrupt disabling failed
FLEXCAN_FREEZE_FAILED	Entering freeze mode failed

16.7 Example

The FlexCAN example application which shows how to use FlexCAN driver API functions is provided with the MQX installation and located in `mqx\examples\can\flexcan` directory.

Chapter 17 NAND Flash Driver

17.1 Overview

This section describes the NAND Flash driver, which is used as an abstraction layer for various Nand Flash Memory devices.

17.2 Source Code Location

Driver	Location
NAND Flash Driver - Generic Part	`source\io\nadflash`
Low Level Code for NAND Flash Controller Module	`source\io\nadflash\nfc`
Low Level Code for SW-driven Implementation	`source\io\nadflash\swdriven`
Parameters of NAND Flash Devices	`source\io\nadflash\nand_devices`

17.3 Header Files

To use NAND Flash driver, include *nandflash.h* and NAND Flash Controller specific header file in your application or BSP (e.g. *nfc.h*).

The *nandflashprv.h* file contains private constants and data structures which NAND Flash drivers use.

17.4 Hardware Supported

The MQX NAND Flash driver currently supports Freescale microprocessors containing NAND Flash Controller (NFC) peripheral module only. However, the driver can be modified to access NAND Flash memory devices directly which is a software driven solution.

MQX NAND Flash driver consists of two layers (see Figure 17-1):

- Lower Layer is platform dependent and has to be customized for particular NFC peripheral (or direct access). This layer implements basic NAND Flash memory operations, and has to provide API described in Section 17.6.1, "NANDFLASH_INIT_STRUCT."
- Upper Layer provides the standard IO functionality (read, write, ioctl …). This layer can be accessed by any MQX application directly, or a file system can be mounted on top of this layer.

User has to describe the structure of the NAND Flash memory to be supported. See Section 17.6.2, "NANDFLASH_INFO_STRUCT". It also has to pass this structure as an initialization parameter during the driver installation. See Section 17.6.1, "NANDFLASH_INIT_STRUCT" for a detailed description.

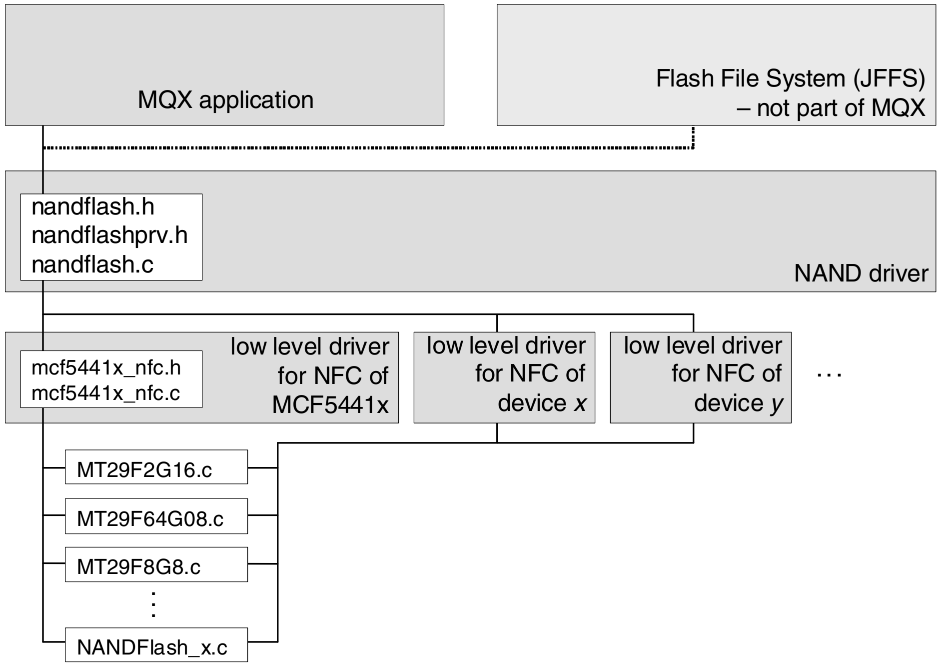


Figure 17-1. MQX NAND Flash Driver Layers

17.5 Driver Services

NAND Flash driver provides the following I/O services.

API	Calls
_io_fopen()	_io_nandflash_open()
_io_fclose()	_io_nandflash_close()
_io_read()	_io_nandflash_read()
_io_write()	_io_nandflash_write()
_io_ioctl()	_io_nandflash_ioctl()

17.6 Installing NAND Flash Driver

NAND Flash driver provides the **_io_nandflash_install()** installation function that either the BSP or the application calls. The function fills in the configuration structures and calls **_io_dev_install_ext()** internally.

In the BSPs distributed with Freescale MQX installation, the **_io_nandflash_install()** installation function is called from *init_bsp.c*. The functionality can be enabled or disabled by setting **BSPCFG_ENABLE_NANDFLASH** configuration option to 1 or 0 in *user_config.h*.

Example

```
result = _io_nandflash_install(&_bsp_nandflash_init);
```

The `_bsp_nandflash_init` is an initialization structure of the NANDFLASH_INIT_STRUCT type containing initialization data for the NAND Flash driver.

17.6.1 NANDFLASH_INIT_STRUCT

This structure contains initialization data and is passed to the NAND Flash driver installation function.

Synopsis

```
struct nandflash_init_struct {
   char_ptr                    ID_PTR;
   uint_32 (_CODE_PTR_         INIT)(struct io_nandflash_struct _PTR_);
   void (_CODE_PTR_            DEINIT)(struct io_nandflash_struct _PTR_);
   uint_32 (_CODE_PTR_         CHIP_ERASE)(struct io_nandflash_struct _PTR_);
   uint_32 (_CODE_PTR_         BLOCK_ERASE)(struct io_nandflash_struct _PTR_,
     uint_32, boolean);
   uint_32 (_CODE_PTR_         PAGE_READ)(struct io_nandflash_struct _PTR_,
     uchar_ptr, uint_32, uint_32);
   uint_32 (_CODE_PTR_         PAGE_PROGRAM)(struct io_nandflash_struct _PTR_,
     uchar_ptr, uint_32, uint_32);
   uint_32 (_CODE_PTR_         WRITE_PROTECT)(struct io_nandflash_struct _PTR_,
     boolean);
   uint_32 (_CODE_PTR_         IS_BLOCK_BAD)(struct io_nandflash_struct _PTR_,
     uint_32);
   uint_32 (_CODE_PTR_         MARK_BLOCK_AS_BAD)(struct io_nandflash_struct _PTR_,
     uint_32);
   _mqx_int (_CODE_PTR_        IOCTL)(IO_NANDFLASH_STRUCT_PTR, _mqx_uint, pointer);
   NANDFLASH_INFO_STRUCT_PTR   NANDFLASH_INFO_PTR;
   _mem_size                   VIRTUAL_PAGE_SIZE;
   _mqx_uint                   NUM_VIRTUAL_PAGES;
   _mqx_uint                   PHY_PAGE_SIZE_TO_VIRTUAL_PAGE_SIZE_RATIO;
   uint_32                     ECC_SIZE;
   _mqx_uint                   WRITE_VERIFY;
   pointer                     DEVICE_SPECIFIC_DATA;
} NANDFLASH_INIT_STRUCT, _PTR_ NANDFLASH_INIT_STRUCT_PTR;
```

Parameters

- ID_PTR — Pointer to a string which identifies the device for **fopen()**.
- INIT— Pointer to the function which initializes the NAND flash device (low-level function).
- DEINIT— Pointer to the function which disables the NAND flash device (low-level function).
- CHIP_ERASE — Pointer to the function which erases the entire NAND flash (low-level function).
- SECTOR_ERASE— Pointer to the function which erases a flash sector (low-level function).
- BLOCK_ERASE— Pointer to the function which erases one NAND flash block (low-level function).
- PAGE_READ — Pointer to the function which reads pages of the NAND flash (low-level function).
- PAGE_PROGRAM — Pointer to the function which programs pages of the NAND flash (low-level function).

- WRITE_PROTECT — Pointer to the function which disables/enables writing to the NAND flash (low-level function).
- IS_BLOCK_BAD — Pointer to the function that checks if the defined block is bad (low-level function).
- MARK_BLOCK_AS_BAD — Pointer to the function which marks the defined block as bad (low-level function).
- IOCTL — Optional function for device specific commands.
- NANDFLASH_INFO_PTR — Pointer to the structure which provides an organization of the NAND flash device. See "NANDFLASH_INFO_STRUCT."
- VIRTUAL_PAGE_SIZE — The size of one virtual page in bytes. One Physical page can be divided into several virtual pages if supported by the NAND Flash Controller. Virtual page is the smallest unit a block device can work with. This value is typically defined in the BSP (BSP_VIRTUAL_PAGE_SIZE).
- NUM_VIRTUAL_PAGES — The number of NAND Flash virtual pages. This value is set by the _io_nandflash_install function.
- PHY_PAGE_SIZE_TO_VIRTUAL_PAGE_SIZE_RATIO — The ratio between the physical page size and the virtual page size. This value is set by the _io_nandflash_install function.
- ECC_SIZE — The number of ECC correction bits per one virtual page. This value is typically defined in the BSP (BSP_ECC_SIZE).
- WRITE_VERIFY — When finished programming, a comparison of data should be made to verify that the write has worked correctly.
- DEVICE_SPECIFIC_DATA — The address of device-specific structure.

Example of nandflash init structure for NFC of MCF5441x device and MT29F2G16 NAND Flash memory:

```
const NANDFLASH_INIT_STRUCT _bsp_nandflash_init =
{
   /* NAME                 */   "nandflash:",
   /* INIT                 */   nfc_init,
   /* DEINIT               */   nfc_deinit,
   /* CHIP_ERASE           */   nfc_erase_flash,
   /* BLOCK_ERASE          */   nfc_erase_block,
   /* PAGE_READ            */   nfc_read_page,
   /* PAGE_PROGRAM         */   nfc_write_page,
   /* WRITE_PROTECT        */   NULL,
   /* IS_BLOCK_BAD         */   nfc_check_block,
   /* MARK_BLOCK_AS_BAD    */   nfc_mark_block_as_bad,
   /* IOCTL                */   nfc_ioctl,
   /* NANDFLASH_INFO_PTR   */   _MT29F2G16_organization_16bit,
   /* VIRTUAL_PAGE_SIZE    */   512,
   /* NUM_VIRTUAL_PAGES    */   0,
   /* PHY_PAGE_SIZE_TO_VIRTUAL_PAGE_SIZE_RATIO */
                                4,
   /* ECC_SIZE             */   4, /* 4-error correction bits (8 ECC bytes) */
   /* WRITE_VERIFY         */   0,
   /* DEVICE_SPECIFIC_DATA */   0
```

```
};
```

All *nfc_xxx* functions are NFC module-dependent low level routines defined in *source/io/nandflash/nfc* subdirectory.

17.6.2 NANDFLASH_INFO_STRUCT

This structure contains information about a particular NAND Flash memory device.

Synopsis

```
struct nandflash_info_struct {
   _mem_size        PHY_PAGE_SIZE;
   _mem_size        SPARE_AREA_SIZE;
   _mem_size        BLOCK_SIZE;
   _mqx_uint        NUM_BLOCKS;
   _mqx_uint        WIDTH;
} NANDFLASH_INFO_STRUCT, _PTR_ NANDFLASH_INFO_STRUCT_PTR;
```

Parameters

- PHY_PAGE_SIZE — The size of the NAND Flash physical page in bytes (without spare bytes).
- SPARE_AREA_SIZE — The size of the NAND Flash spare area in bytes.
- SPARE_AREA_SIZE — The size of one block in bytes.
- NUM_BLOCKS — The number of NAND Flash blocks.
- WIDTH — The width of the device in bytes.

Example of nandflash info structure for MT29F2G16 NAND Flash memory:

```
#define MT29F2G16_PHYSICAL_PAGE_SIZE      2048
#define MT29F2G16_SPARE_AREA_SIZE         64
#define MT29F2G16_BLOCK_SIZE              131072 /* 128kB */
#define MT29F2G16_NUM_BLOCKS              2048
#define MT29F2G16_WIDTH                   16

NANDFLASH_INFO_STRUCT _MT29F2G16_organization_16bit[] =  {
  MT29F2G16_PHYSICAL_PAGE_SIZE,
  MT29F2G16_SPARE_AREA_SIZE,
  MT29F2G16_BLOCK_SIZE,
  MT29F2G16_NUM_BLOCKS,
  MT29F2G16_WIDTH
};
```

17.7 NFC Peripheral Module-Specific Low Level Routines

NAND Flash driver refers to low-level functions which implement NAND flash atomic operations. These functions are part of the MQX release for all supported NFCs. The user passes pointers to these low-level functions in the `NANDFLASH_INIT_STRUCT` when installing the NAND Flash driver.

The functions are located in NFC-specific subdirectory in *source/io/nandflash/nfc*.

17.7.1 Init Function

This function initializes the NAND flash device.

Synopsis

```
uint_32 (_CODE_PTR_  INIT)(
        IO_NANDFLASH_STRUCT_PTR nandflash_ptr)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.

17.7.2 De-init Function

This function de-initializes the NAND flash device.

Synopsis

```
void (_CODE_PTR_  DEINIT)(
        IO_NANDFLASH_STRUCT_PTR nandflash_ptr)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.

17.7.3 Chip Erase Function

This function erases the entire NAND flash device.

Synopsis

```
uint_32 (_CODE_PTR_ CHIP_ERASE)(
        IO_NANDFLASH_STRUCT_PTR nandflash_ptr)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.

17.7.4 Block Erase Function

This function erases one NAND flash block.

Synopsis

```
uint_32 (_CODE_PTR_ BLOCK_ERASE)(
        IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
        uint_32                 block_number,
        boolean                 force_flag)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.
- *block_number [IN]* — Number of block to erase.
- *force_flag [IN]*
  — TRUE to force block erase in case the block is marked as bad.
  — FALSE if there is no need to force block erase.

17.7.5 Page Read Function

This function reads pages of the NAND flash.

Synopsis

```
uint_32 (_CODE_PTR_ PAGE_READ)(
        IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
        uchar_ptr               to_ptr,
        uint_32                 page_number,
        uint_32                 page_count)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.
- *to_ptr [OUT]* — Where to copy the data to.
- *page_number [IN]* — Page number where to start reading.
- *page_count [IN]* — The amount of pages to be read.

17.7.6 Page Program Function

This function programs the pages of the NAND flash.

Synopsis

```
uint_32 (_CODE_PTR_ PAGE_PROGRAM)(
        IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
        uchar_ptr               from_ptr,
        uint_32                 page_number,
        uint_32                 page_count)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.
- *from_ptr [IN]* — Where to copy the data from.
- *page_number [IN]* — Page number where to start writing.
- *page_count [IN]* — The number of pages to be programed.

17.7.7 Write Protect Function

This function is optional. This function is called to write-enable or write-protect the device.

Synopsis

```
uint_32 (_CODE_PTR_ WRITE_PROTECT)(
        IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
        boolean                 write_protect)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.
- *write_protect [IN]*
  - — TRUE if the device is to be write-protected.
  - — FALSE to allow writing to the device.

17.7.8 Is Block Bad Function

This function checks if the defined block is bad.

Synopsis

```
uint_32 (_CODE_PTR_ IS_BLOCK_BAD)(
        IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
        uint_32                 block_number)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.
- *block_number [IN]* — The block number to be checked.

17.7.9 Mark Block as Bad Function

This function is called to mark the defined block as bad.

Synopsis

```
uint_32 (_CODE_PTR_ MARK_BLOCK_AS_BAD)(
        IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
        uint_32                 block_number)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.
- *block_number [IN]* — The block number to be marked as bad.

17.8 I/O Control Commands

This section describes the I/O control commands that can be used when **_io_ioctl()** is called. Commands are defined in *nandflash.h*.

Command	Description
NANDFLASH_IOCTL_GET_PHY_PAGE_SIZE	Gets the NAND Flash physical page size.
NANDFLASH_IOCTL_GET_SPARE_AREA_SIZE	Gets the NAND Flash spare area size.
NANDFLASH_IOCTL_GET_BLOCK_SIZE	Gets the NAND Flash block size.
NANDFLASH_IOCTL_GET_NUM_BLOCKS	Gets the total number of NAND Flash blocks.
NANDFLASH_IOCTL_GET_WIDTH	Gets the NAND Flash width.
NANDFLASH_IOCTL_GET_NUM_VIRT_PAGES	Gets the total number of virtual pages.
NANDFLASH_IOCTL_GET_VIRT_PAGE_SIZE	Gets the size of one virtual page.
NANDFLASH_IOCTL_ERASE_BLOCK	Erases the specified block of the NAND Flash.
NANDFLASH_IOCTL_ERASE_CHIP	Erases the whole NAND Flash.
NANDFLASH_IOCTL_WRITE_PROTECT	Write-enable or write-protect the NAND Flash device.
NANDFLASH_IOCTL_GET_WRITE_PROTECT	Returns 1 if the flash is write-protected, otherwise it returns 0.
NANDFLASH_IOCTL_CHECK_BLOCK	Checks if the defined NAND Flash block is bad or not.
NANDFLASH_IOCTL_MARK_BLOCK_AS_BAD	Marks the defined NAND Flash block as bad.
NANDFLASH_IOCTL_GET_BAD_BLOCK_TABLE	Checks all NAND Flash blocks and gets the bad block table (field of 8-bit values, length equals to the number of NAND Flash blocks, 0 = bad block, 1 = not a bad block).
NANDFLASH_IOCTL_GET_ID	Gets NAND Flash ID.
NANDFLASH_IOCTL_ERASE_BLOCK_FORCE	Forces block erase in case the block is marked as bad.

17.9 Example

The NAND Flash example application that shows how to use NAND Flash driver is provided with the MQX installation and is located in `mqx\examples\nandflash` directory.

17.10 Error Codes

This section describes all error codes that can be returned by the NAND Flash driver. Error codes are defined in *nandflash.h*.

Error Code	Description
NANDFLASHERR_NO_ERROR	Operation successful.
NANDFLASHERR_ECC_FAILED	Returned when the ECC engine finds that the read page cannot be corrected.
NANDFLASHERR_ECC_CORRECTED	Returned when the ECC engine corrected errors is the read page.
NANDFLASHERR_ERASE_FAILED	Returned when the erasing process fails.
NANDFLASHERR_WRITE_FAILED	Returned when writing to the NAND Flash fails.
NANDFLASHERR_TIMEOUT	Returned when any operation with the NAND Flash is timed-out.
NANDFLASHERR_BLOCK_BAD	Returned when the specified block is bad.
NANDFLASHERR_BLOCK_NOT_BAD	Returned when the specified block is not bad.
NANDFLASHERR_INFO_STRUC_MISSING	Returned when the NANDFLASH_INFO_STRUCT is not available for the driver (not defined manually and simultaneously not possible to create from the NAND ID read out of the NAND Flash).
NANDFLASHERR_IMPROPER_ECC_SIZE	Returned when the sum of virtual page size (incl. ECC bytes) per one physical page is not greater than the physical page size plus the number of physical spare bytes.

Chapter 18 DAC Driver

18.1 Overview

This section describes Digital to Analog Converter (DAC) driver that accompanies the MQX release.

DAC driver implements custom API and does not follow the standard driver interface (I/O Subsystem). The driver code is separated into Logical Device Driver (LDD) layer and Physical Device Driver (PDD) layer. This driver structure is adopted from the new Processor Expert component technology which is available for Freescale Semiconductor platforms.

18.2 Source Code Location

The source files for the DAC driver are located in `source\io\dac` directory.

18.3 Header Files

To use the DAC driver with the DAC peripheral module, include the header file *bsp.h* into your application. The *bsp.h* file includes all DAC header files.

18.4 API Function Reference

This section serves as a function reference for the DAC module(s).

18.4.1 DAC_Init()

This function (re)initializes the DAC module.

Synopsis

```
LDD_TDeviceDataPtr DAC_Init (
    /* [IN] Pointer to the RTOS device structure. */
    LDD_RTOS_TDeviceDataPtr     RTOSDeviceData
);
```

Parameters

`RTOSDeviceData` *[in]* — Pointer to the private device structure. This pointer is passed to all callback events as parameter.

Description

Initializes the device according to the design time configuration properties. Allocates memory for the device data structure. This method can be called only once. Before the second call of DAC_Init(), the DAC_Deinit() must be called first.

Return Value

`LDD_TDeviceDataPtr` — Pointer to the dynamically allocated private structure or NULL if there was an error.

Example

The following example shows how to initialize the DAC module.

```
/* DAC callback function prototypes */
void DAC_BufferStartCallBack(LDD_RTOS_TDeviceDataPtr DeviceData);
void DAC_BufferWattermarkCallBack(LDD_RTOS_TDeviceDataPtr DeviceData);
void DAC_BufferEndCallBack(LDD_RTOS_TDeviceDataPtr DeviceData);

/* DAC init structure */
const LDD_RTOS_TDeviceData  DAC_RTOS_DeviceData =
{
/* DAC device number                */  DAC_1,
/* DAC reference selection          */  DAC_PDD_V_REF_EXT,
/* DAC trigger mode                 */  DAC_PDD_HW_TRIGGER,
/* DAC buffer mode                  */  LDD_DAC_BUFFER_NORMAL_MODE,
/* DAC buffer start callback        */  DAC_BufferStartCallBack,
/* DAC buffer watermark callback    */  DAC_BufferWattermarkCallBack,
/* DAC buffer end callback          */  DAC_BufferEndCallBack
};

/* Initialize DAC device */
if (NULL == (DAC_DevicePtr = DAC_Init((const
LDD_RTOS_TDeviceDataPtr)&DAC_RTOS_DeviceData)))
{
```

```
        printf("DAC device initialization failed\n");
}
```

18.4.2 DAC_Deinit()

The function deinitializes DAC device.

Synopsis

```
void DAC_Deinit (
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr DeviceData
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.

Description

Disables the device and releases the device data structure memory.

Return Value

- none

18.4.3 DAC_Enable()

This function enables the DAC device.

Synopsis

```
LDD_TError DAC_Enable (
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr DeviceData
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.

Description

Enables the DAC device. If possible, this method switches on digital-to-analog converter device, voltage reference, etc. This method is intended to be used together with DAC_Disable method to temporarily switch On/Off the device after the device is initialized.

Return Value

- DAC_ERROR_OK (success)

Example

The following example enables the DAC device initialized in the DAC_Init() example code

```
printf ("Enabling DAC device... ");
if (DAC_ERROR_OK != DAC_Enable(DAC_DevicePtr)) {
        printf ("Error!\n");
}
```

18.4.4 DAC_Disable()

This function disables the DAC device.

Synopsis

```
LDD_TError DAC_Disable  (
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr DeviceData
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.

Description

Disables the DAC device. If possible, this method switches off digital-to-analog converter device, voltage reference, etc. This method is intended to be used together with DAC_Enable method to temporarily switch On/Off the device after the device is initialized. This method is not required. The Deinit() method can be used to switch off and uninstall the device.

Return Value

- DAC_ERROR_OK – OK

Example

The following example disables the DAC device:

```
DAC_Disable(DAC_DevicePtr);
```

18.4.5 DAC_SetEventMask()

This function enables the DAC callback events

Synopsis

```
LDD_TError DAC_SetEventMask (
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr DeviceData,
    /* [IN] Mask of events to enable. */
    LDD_TEventMask EventMask
)
```

Parameters

`DeviceData [in]` – Device data structure pointer.
`EventMask [in]` – Mask of events to enable.

Description

Enables/disables event(s). This method is available if the interrupt service/event property is enabled and at least one event is enabled. Pair method to GetEventMask().

Return Value

- DAC_ERROR_OK – OK
- DAC_ERROR_VALUE – Event mask is not valid.
- DAC_ERROR_DISABLED – This component is disabled by user.

Example

The following example shows how to enable the DAC buffer watermark and buffer end events.

```
DAC_Error = DAC_SetEventMask(DAC_DevicePtr, (LDD_DAC_ON_BUFFER_WATERMARK |
LDD_DAC_ON_BUFFER_END));

switch (DAC_Error)
{
    case DAC_ERROR_OK:
    /* OK */
    break;
    case DAC_ERROR_VALUE :
    case DAC_ERROR_DISABLED :
    /* Wrong mask or device disabled error */
    break;
}
```

18.4.6 DAC_GetEventMask()

This function returns the current masks of enabled events.

Synopsis

```
LDD_TEventMask DAC_GetEventMask (
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr DeviceData
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.

Description

Returns the current events mask. This method is available if the interrupt service/event property is enabled and at least one event is enabled. Pair method to SetEventMask().

Return Value

- *LDD_TEventMask* — Mask of enabled events.

18.4.7 DAC_GetEventStatus()

This function returns the state of DAC status flags.

Synopsis

```
LDD_TEventMask DAC_GetEventStatus (
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr DeviceData
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.

Description

This method returns the current state of the status flags and clears the pending interrupt flags. Return value has the same format as EventMask parameter of SetEventMask() method. Can be used for polling mode without using events.

Return Value

- `LDD_TEventMask` – Current mask of pending events.

Example

The following example shows how to handle the DAC device in polling mode.

```
/* DAC RTOS init structure - no interrupt callbacks are installed */
const LDD_RTOS_TDeviceData  DAC_RTOS_DeviceData =
{
/* DAC device number                */  DAC_1,
/* DAC reference selection          */  DAC_PDD_V_REF_EXT,
/* DAC trigger mode                 */  DAC_PDD_HW_TRIGGER,
/* DAC buffer mode                  */  LDD_DAC_BUFFER_NORMAL_MODE,
/* DAC buffer start callback        */  NULL,
/* DAC buffer watermark callback    */  NULL,
/* DAC buffer end callback          */  NULL
};

/* Global DAC variables */
LDD_TDeviceDataPtr DAC_DevicePtr;
LDD_TEventMask DAC_EventMask;

/* Initialize DAC device for polling mode */
DAC_DevicePtr = DAC_Init((const LDD_RTOS_TDeviceDataPtr)&DAC_RTOS_DeviceData));
if (NULL == DAC_DevicePtr) {
        printf("DAC device initialization failed\n");
}

printf ("Enabling DAC device... ");
if (DAC_ERROR_OK != DAC_Enable(DAC_DevicePtr)) {
        printf ("Error!\n");
}
/* in some periodically called function poll event status and handle buffer */
```

```
DAC_EventMask = DAC_GetEventStatus (DAC_DeviceData);
switch (DAC_EventMask)
{
    case LDD_DAC_ON_BUFFER_START:
        /* buffer start*/
        DAC_Error = DAC_SetBuffer(...);
    break;

    case LDD_DAC_ON_BUFFER_WATERMARK:
        /* watermark reached */
        DAC_Error = DAC_SetBuffer(...);

    break;

    case LDD_DAC_ON_BUFFER_END:
        /* buffer is empty */
        DAC_Error = DAC_SetBuffer(...);
    break;
}
```

18.4.8 DAC_SetValue()

This function sets the DAC output value.

Synopsis

```
LDD_TError DAC_SetValue (
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr  DeviceData,
    /* [IN] User data */
    LDD_DAC_TData       Data
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.
`Data [in]` – Device data structure pointer.

Description

Sets the DAC output voltage to the specified value. This method is used when data buffering is not required. The 12-bit right justified format is assumed and no data transformation (shifting or scaling) is done in the driver.

Return Value

- DAC_ERROR_OK – OK

Example

The following example shows how to set DC value on the DAC device.

```
DAC_Error = DAC_SetValue (DAC_DevicePtr, (LDD_DAC_TData)2048);
```

18.4.9 DAC_SetBuffer()

This function writes data from the user buffer to the DAC buffer.

Synopsis

```
LDD_TError DAC_SetBuffer (
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr  DeviceData,
    /* [IN] Pointer to array containing user data. */
    uint_16_ptr         DataArrayPtr,
    /* [IN] Length of user data array which should be written to data buffer. */
    uint_8              DataArrayLength,
    /* [IN] Index of first written data buffer register. */
    uint_8              StartBufferReg
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.
`DataArrayPtr [in]` – Pointer to array containing user data.
`DataArrayLength [in]` – Length of user data array which should be written to data buffer.
`StartBufferReg [in]` – Index of first written data buffer register.

Description

Writes an array of data words to the data buffer registers. Array is defined by a pointer to the start address and by length. First written data buffer register is defined by an index. The rest of the array is written to registers with increasing index. If the length of array exceeds number of registers between the first written register and the last register at the end of the buffer, then DAC_ERROR_RANGE is returned and no data is written.

It is possible to write all registers available in the hardware. The check for the current upper limit value of the buffer is not done. Therefore, it is possible to write data to the whole data buffer regardless of the current configuration.

DataArrayPtr has the fixed data type regardless of the current hardware or design time configuration and must always be used.

Return Value

- DAC_ERROR_OK – OK
- DAC_ERROR_RANGE – Parameter out of range

Example

The following example shows how to write to the DAC device buffer.

```
#define DAC_INTERNAL_BUFFER_SIZE     16

... variable definition section
static uint_16     DAC_BufferWaterMark = LDD_DAC_BUFFER_WATERMARK_L4;
static uint_16_ptr GEN_BufferPtr;
```

```
... code in some function
... initialize GEN_BufferPtr

/* Set Buffer Watermark */
DAC_Error = DAC_SetBufferWatermark(DAC_DevicePtr, DAC_BufferWaterMark);

/* Copy data from buffer start to watermark */
DAC_Error = DAC_SetBuffer(
        DAC_DevicePtr,
        GEN_BufferPtr,
        DAC_INTERNAL_BUFFER_SIZE - DAC_BufferWaterMark - 1,
        0
);

/* Increment buffer pointer */
GEN_BufferPtr += (DAC_INTERNAL_BUFFER_SIZE - DAC_BufferWaterMark - 1);
```

18.4.10 DAC_SetBufferReadPointer()

This function sets the DAC internal buffer read pointer.

Synopsis

```
LDD_TError DAC_SetBufferReadPointer(
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr  DeviceData,
    /* [IN] New read pointer value. */
    uint_8              Pointer
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.
`Pointer [in]` – New read pointer value.

Description

Sets the data buffer read pointer value. If requested pointer value is greater than buffer size defined by buffer upper limit value, then error is returned.

Return Value

- DAC_ERROR_OK – OK
- DAC_ERROR_RANGE – Pointer value out of range

Example

The following example shows how to set the DAC buffer read pointer:

```
/* Set buffer read pointer to buffer start */
DAC_Error = DAC_SetBufferReadPointer(
        DAC_DevicePtr,
        0
);
```

18.4.11 DAC_SetBufferMode()

This function sets the DAC internal buffer mode.

Synopsis

```
LDD_TError DAC_SetBufferMode(
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr  DeviceData,
    /* [IN] - Buffer work mode. */
    LDD_DAC_TBufferMode Mode
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.
`Mode [in]` – Buffer work mode.

Description

Selects the buffer work mode.

- LDD_DAC_BUFFER_DISABLED – Buffer Mode Disabled
- LDD_DAC_BUFFER_NORMAL_MODE – Buffer Normal Mode

  This is the default mode. The buffer works as a circular buffer. The read pointer increases by one every time when the trigger occurs. When the read pointer reaches the upper limit, it goes directly to zero in the next trigger event.
- LDD_DAC_BUFFER_SWING_MODE – Buffer Swing Mode

  This mode is similar to the Normal mode. However, when the read pointer reaches the upper limit, it does not go to the zero. It will descend by one in the next trigger event until zero is reached.
- LDD_DAC_BUFFER_OTSCAN_MODE – One-time scan mode

  The read pointer increases by one every time the trigger occurs. When it reaches the upper limit, it stops. If the read pointer is reset to an address other than the upper limit, it will increase to the upper address and then stop.

Return Value

- DAC_ERROR_OK – OK

Example

The following example shows how to set the DAC buffer read pointer:

```
/* Set DAC internal buffer to circular mode */
DAC_Error = DAC_SetBufferMode(
        DAC_DevicePtr,
        LDD_DAC_BUFFER_NORMAL_MODE
);
```

18.4.12 DAC_SetBufferReadPointer()

This function sets the DAC internal buffer read pointer.

Synopsis

```
LDD_TError DAC_SetBufferReadPointer(
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr  DeviceData,
    /* [IN] New read pointer value. */
    uint_8              Pointer
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.
`Pointer [in]` – New read pointer value.

Description

Sets the data buffer read pointer value. If the requested pointer value is greater than buffer size defined by buffer upper limit value, then error is returned.

Return Value

- DAC_ERROR_OK – OK
- DAC_ERROR_RANGE – Pointer value out of range.

Example

The following example shows how to set the DAC buffer read pointer:

```
/* Set buffer read pointer to buffer start */
DAC_Error = DAC_SetBufferReadPointer(
        DAC_DevicePtr,
        0
);
```

18.4.13 DAC_SetBufferSize()

This function sets the DAC internal buffer size.

Synopsis

```
LDD_TError DAC_SetBufferSize(
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr  DeviceData,
    /* [IN] Number of data buffer registers. */
    uint_8              Size
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.
`Watermark [in]` – Number of words between the read pointer and upper address.

Description

Sets the data buffer size. If requested buffer size exceeds hardware capacity then DAC_ERROR_RANGE is returned.

Return Value

- DAC_ERROR_OK – OK
- DAC_ERROR_RANGE – Requested buffer size out of range.

Example

The following example shows how to set the DAC buffer size.

```
/* Set DAC internal buffer size to 16 words (max. value)*/
DAC_Error = DAC_SetBufferSize(
        DAC_DevicePtr,
        16
);
```

18.4.14 DAC_ForceSWTrigger()

This function triggers internal data buffer read pointer.

Synopsis

```
LDD_TError DAC_ForceSWTrigger(
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr  DeviceData
);
```

Parameters

`DeviceData [in]` – Device data structure pointer.

Description

Trigger internal buffer read pointer.

Return Value

- DAC_ERROR_OK – OK
- DAC_ERROR_DISABLED – HW trigger is selected or buffer is disabled.

Example

The following example shows how to set the DAC buffer size.

```
/* Set DAC internal buffer size to 16 words (max. value)*/
DAC_Error = DAC_SetBufferSize(
        DAC_DevicePtr,
        16
);
```

18.5 Data Types Used by the DAC Driver API

18.5.1 LDD_TDeviceDataPtr

A pointer to a 32-bit unsigned integer and to the private structure containing component state information. Init method of the component creates the private state structure and returns the pointer to it. This pointer needs to be passed to every component method.

Definition

```
typedef pointer LDD_TDeviceDataPtr;
```

18.5.2 LDD_RTOS_TDeviceDataPtr

A pointer to the structure used by RTOS containing driver-specific information. Init method receives this pointer and then passes it to all events and call-backs.

Definition

```
typedef struct
{
    /* DAC device number */
    uint_8        DAC_DEVICE_NUMBER;
    /* DAC reference selection */
    uint_8        DAC_REFSEL;
    /* DAC trigger mode */
    uint_8        DAC_TRIGER_MODE;
    /* DAC buffer mode */
    uint_8        DAC_MODE;
    /* DAC start buffer callback */
    void (_CODE_PTR_ DAC_PDD_BUFFER_START_CALLBACK) (LDD_RTOS_TDeviceDataPtr);
    /* DAC start buffer callback */
    void (_CODE_PTR_ DAC_PDD_BUFFER_WATERMARK_CALLBACK)(LDD_RTOS_TDeviceDataPtr);
    /* DAC end buffer callback */
   void (_CODE_PTR_ DAC_PDD_BUFFER_END_CALLBACK) (LDD_RTOS_TDeviceDataPtr);
} LDD_RTOS_TDeviceData, _PTR_ LDD_RTOS_TDeviceDataPtr;
```

- DAC_DEVICE_NUMBER – The number of device to initialize. The MCF51MM has only 1 DAC device to use DAC_1.
- DAC_REFSEL – DAC device reference selection. The DAC device on MCF51MM supports two references. Use DAC_PDD_V_REF_INT for internal reference or DAC_PDD_V_REF_EXT for external VREF.
- DAC_TRIGER_MODE – Select trigger mode. Use DAC_PDD_HW_TRIGGER for hardware triggering by Programmable Delay Block (PDB) or DAC_PDD_SW_TRIGGER for software triggering using DAC_ForceSWTrigger() method.
- DAC_MODE – DAC buffering mode. Use LDD_DAC_BUFFER_DISABLED or LDD_DAC_BUFFER_NORMAL_MODE or LDD_DAC_BUFFER_SWING_MODE or LDD_DAC_BUFFER_OTSCAN_MODE.
- DAC_PDD_BUFFER_START_CALLBACK – Specify the name of DAC Start Buffer Callback. If NULL is specified, no callback is installed and start buffer interrupt is disabled.
- DAC_PDD_BUFFER_WATERMARK_CALLBACK – Specify the name of DAC Watermark Buffer Callback. If NULL is specified, no callback is installed and watermark buffer interrupt is disabled.
- DAC_PDD_BUFFER_END_CALLBACK – Specify the name of DAC end Buffer Callback. If NULL is specified no callback is installed and end buffer interrupt is disabled.

18.5.3 LDD_DAC_TBufferMode

This data type is intended to be used for declaration of DAC data buffer work modes that will be passed to SetBufferMode method.

Definition

```
typedef enum {
    LDD_DAC_BUFFER_DISABLED     = 0,
    LDD_DAC_BUFFER_NORMAL_MODE  = 1,
    LDD_DAC_BUFFER_SWING_MODE   = 2,
    LDD_DAC_BUFFER_OTSCAN_MODE  = 3
} LDD_DAC_TBufferMode;
```

18.5.4 LDD_DAC_TBufferWatermark

This data type is intended to be used for the declaration of DAC data buffer watermark levels that will be passed to SetBufferWatermark methods.

Definition

```
typedef enum {
    LDD_DAC_BUFFER_WATERMARK_L1 = 0,     /* 1 word  */
    LDD_DAC_BUFFER_WATERMARK_L2 = 1,     /* 2 words */
    LDD_DAC_BUFFER_WATERMARK_L3 = 2,     /* 3 words */
    LDD_DAC_BUFFER_WATERMARK_L4 = 3      /* 4 words */
} LDD_DAC_TBufferWatermark;
```

18.5.5 LDD_DAC_TData

A 32-bit unsigned integer user data type. This data type is intended to be used for declaration of data which is passed to the set data register methods. The size of this data type is always maximum regardless of the current design time configuration, and may vary only across the different MCU families.

Definition

```
typedef uint_32  LDD_DAC_TData;
```

18.5.6 LDD_TEventMask

DAC event mask type specified in the *dac_ldd.h* header file. It is used by DAC_SetEventMask(), DAC_GetEventMask(), and DAC_GetEventStatus() functions.

Definition

```
typedef uint_32 LDD_TEventMask;
```

DAC driver supports the following error values:

- LDD_DAC_ON_BUFFER_START – Internal DAC buffer read pointer reached buffer start.
- LDD_DAC_ON_BUFFER_WATERMARK – Internal DAC buffer read pointer reached watermark level.
- LDD_DAC_ON_BUFFER_END – Internal DAC buffer read pointer reached buffer end.

18.6 Example

The DAC example application that shows how to generate 1 kHz sine signal using DAC Normal buffering mode. The DAC driver API functions are provided with the MQX installation and located in `mqx\examples\dac` directory.

18.7 Error Codes

18.7.1 LDD_TError

Error identifier type specified in the *dac_ldd.h* header file. It is used to return error values.

Synopsis

```
typedef uint_16 LDD_TError;
```

DAC driver supports the following error values:

- DAC_ERROR_OK – No Error.
- DAC_ERROR_DISABLED – DAC device is disabled by user.
- DAC_ERROR_VALUE – Value is not valid.
- DAC_ERROR_RANGE – Parameter out of range.

Chapter 19 LWGPIO Driver

19.1 Overview

This section describes the Light-Weight GPIO (LWGPIO) driver that accompanies the MQX. This driver is a common interface for GPIO modules.

The LWGPIO driver implements custom API and does not follow the standard driver interface (I/O Subsystem). Therefore, it can be used before the I/O subsystem of the MQX is initialized. LWGPIO driver is designed as a per-pin driver, meaning that LWGPIO API call handles only one pin.

19.2 Source Code Location

The source files for the LWGPIO driver are located in `source\io\lwgpio` directory. *lwgpio_* file prefix is used for all LWGPIO module related API files.

19.3 Header Files

To use the LWGPIO driver, include the *lwgpio.h* header file and the platform specific header file, *lwgpio_mcf52xx.h*, into your application or into the BSP header file, *bsp.h*. The platform specific header file should be included before *lwgpio.h*.

Header file for Kinetis platforms is called *lwgpio_kgpio.h*.

19.4 API Function Reference

This sections serves as a function reference for the LWGPIO module(s).

This function sets a property of the pin. For example a pull up resistor, a pull down resistor, drive strength, slew-rate, filters etc.

19.4.1 lwgpio_init()

This function initializes the structure for a GPIO pin that will be used as a pin handle in the other API functions of the LWGPIO driver. It also performs basic GPIO register pre-initialization.

Synopsis

```
boolean lwgpio_init
(
    LWGPIO_STRUCT_PTR handle,
    LWGPIO_PIN_ID     id,
    LWGPIO_DIR        dir,
    LWGPIO_VALUE      value
)
```

Parameters

handle [in/out] — Pointer to the LWGPIO_STRUCT structure that will be filled in.
id [in] — LWGPIO_PIN_ID number identifying pin (platform and peripheral specific).
dir [in] — LWGPIO_DIR enum value for initial direction control.
value [in] — LWGPIO_VALUE enum value for initial output control.

Description

The *lwgpio_init()* function has to be called prior to calling any other API function of the LWGPIO driver. This function initializes the LWGPIO_STRUCT structure. The pointer to the LWGPIO_STRUCT is passed as a *handle* parameter. To identify the pin, platform-specific LWGPIO_PIN_ID number is used.

The variable *dir* of type LWGPIO_DIR can have the following values:

- LWGPIO_DIR_INPUT - Presets pin into input state.
- LWGPIO_DIR_OUTPUT - Presets pin into output state.
- LWGPIO_DIR_NOCHANGE - Does not preset pin into input/output state.

The variable *value* of type LWGPIO_VALUE can have the following values:

- LWGPIO_VALUE_LOW - Presets pin into active low state.
- LWGPIO_VALUE_HIGH - Presets pin into active high state.
- LWGPIO_VALUE_NOCHANGE - Does not preset pin into low/high state.

If the *value* is set to LWGPIO_VALUE_LOW or LWGPIO_VALUE_HIGH and the *dir* parameter is passed as a LWGPIO_DIR_OUTPUT, the corresponding level is set on the GPIO output latch, if at all possible and depending on a peripheral, and the pin is set to the output state. This function does not configure the GPIO mode of the pin.

Return Value

- TRUE (Success)
- FALSE (Failure)

Example

The following example shows how to initialize the LWGPIO pin PTA-3 on MCF52259 MCU.

```
LWGPIO_STRUCT led1;
status = lwgpio_init(&led1,
                         LWGPIO_PORT_TA | LWGPIO_PIN3,
                         LWGPIO_DIR_OUTPUT,
                         LWGPIO_VALUE_HIGH);
if (status != TRUE)
{
        printf("Initializing GPIO as output failed.\n");
        _mqx_exit(-1);
```

```
}
```

19.4.2 lwgpio_set_functionality()

This function sets the functionality of the pin.

Synopsis

```
void lwgpio_set_functionality
(
    LWGPIO_STRUCT_PTR handle,
    uint_32           functionality
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by the lwgpio_init() function.
functionality [in] — An integer value which represents the requested functionality of the GPIO pin. This is a HW-dependent constant.

Description

This function allows assigning the requested functionality to the pin for the GPIO mode or any other peripheral mode. The value of the *functionality* parameter represents the number stored in the multiplexer register field which selects the desired functionality. For the GPIO mode, you can use the pre-defined macros which can be found in the *lwgpio_ <mcu>.h* file.

Return Value

- None

Example

The following example shows how to set LWGPIO pin PTA.3 on MCF52259 MCU in the GPIO peripheral mode.

```
lwgpio_set_functionality(&led1, LWGPIO_MUX_PTA3_GPIO);
```

19.4.3 lwgpio_get_functionality()

This function gets the actual peripheral functionality of the pin. The pin peripheral function mode depends on the MCU.

Synopsis

```
uint_32 lwgpio_get_functionality
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by lwgpio_init() function.

Description

This function is the inverse of the lwgpio_set_functionality(). It returns a value stored in the multiplexer register field which defines the desired functionality.

Return Value

- An integer value representing the actual pin functionality.

Example

The following example shows how to get functionality for a pin on MCF52259 MCU.

```
func = lwgpio_get_functionality(&led1);
```

19.4.4 lwgpio_set_direction()

This function sets direction (input or output) of the specified pin.

Synopsis

```
void lwgpio_set_direction
(
    LWGPIO_STRUCT_PTR handle,
    LWGPIO_DIR        dir
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by the lwgpio_init() function.
dir [in] — One of the LWGPIO_DIR enum values.

Description

This function is used to change the direction of the specified pin. As this function does not change the pin's functionality, it is possible to set the direction of a pin that is currently not in the GPIO mode.

Return Value

- None

Example

The following example shows how to set the LWGPIO pin direction to the output on MCF52259.

```
lwgpio_set_direction(&led1, LWGPIO_DIR_OUTPUT);
```

19.4.5 lwgpio_set_value()

This function sets the pin state (low or high) of the specified pin.

Synopsis

```
void lwgpio_set_value
(
    LWGPIO_STRUCT_PTR handle,
    LWGPIO_VALUE      value
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by the lwgpio_init() function.
value [in] — One of the LWGPIO_VALUE enum values.

Description

This function is used to change the specified pin state. As this function does not change either the pin's functionality or the direction, it is possible to set the pin state of a pin that is currently not in the GPIO mode. Similarly, it is possible to set the pin state of a pin that is set for input direction and have it ready for future changing of the pin direction.

Return Value

- None

Example

The following example shows how to set the pin state as "high" for the LWGPIO pin on MCF52259.

```
lwgpio_set_value(&led1, LWGPIO_VALUE_HIGH);
```

19.4.6 lwgpio_toggle_value()

This function toggles the pin state (low or high) of the specified pin.

Synopsis

```
void lwgpio_toggle_value
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by the lwgpio_init() function.

Description

This function is used for changing (toggling) the specified pin state.

Return Value

- none

Example

The following example shows how to toggle the pin state for the LWGPIO pin on MCF52259.

```
lwgpio_toggle_value(&led1);
```

19.4.7 lwgpio_get_value()

This function gets voltage value (low or high) of the specified pin.

Synopsis

```
LWGPIO_VALUE lwgpio_get_value
(
```

```
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by the lwgpio_init() function.

Description

This function is the inverse of the lwgpio_set_value() function. The direct relation between the physical pin state and the result of this function does not always exist, because this function gets the output buffer value rather than sampling pin voltage level of a pin that is set to output. To sample the pin voltage level, use lwgpio_get_raw() function. If the GPIO functionality is not assigned to the pin, the result of this function is not specified.

Return Value

- LWGPIO_VALUE - voltage value of the specified pin

Example

The following example shows how to get voltage level for the LWGPIO pin on MCF52259.

```
LWGPIO_VALUE value = lwgpio_get_value(&button1);
```

19.4.8 lwgpio_get_raw()

This function gets raw voltage value (low or high) of the specified pin if supported by target MCU.

Synopsis

```
LWGPIO_VALUE lwgpio_get_raw
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by the lwgpio_init() function.

Description

This function samples the pin signal to get the voltage value. If the GPIO functionality is not assigned to the pin, the result of this function is not specified.

Return Value

- LWGPIO_VALUE - Voltage value of the specified pin

Example

The following example shows how to get the physical voltage level for the LWGPIO pin on MCF52259.

```
LWGPIO_VALUE value = lwgpio_get_raw(&button1);
```

19.4.9 lwgpio_int_init()

This function initializes interrupt for the specified pin.

Synopsis

```
boolean lwgpio_int_init
(
    LWGPIO_STRUCT_PTR handle,
    LWGPIO_INT_MODE   mode
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by lwgpio_init() function.
mode [in] — Value consisting of a logical combination of the LWGPIO_INT_xxx flags.

Description

This function prepares the pin for the interrupt mode. It configures the interrupt peripheral to generate the interrupt flag. For most platforms, this function does not enable interrupts and it does not modify the GPIO peripheral settings. If there is a need to turn a pin into a GPIO functionality in order to get the interrupt running, the user must do it manually prior to calling the lwgpio_int_init() function. In general, it is recommended to set the pin to the GPIO input state prior to the interrupt initialization.

Return Value

- TRUE (Success)
- FALSE (Failure)

Example

The following example shows how to initialize the rising edge interrupt for the LWGPIO pin PNQ.3 on MCF52259.

```
status = lwgpio_init(
    &btn_int,
    LWGPIO_PORT_NQ | LWGPIO_PIN3,
    LWGPIO_DIR_INPUT,
    LWGPIO_VALUE_NOCHANGE);

if (status == TRUE)
{
    status = lwgpio_int_init(&btn_int, LWGPIO_INT_MODE_RISING);
}

if (status != TRUE)
{
    printf("Initializing pin for interrupt failed.\n");
    _mqx_exit(-1);
}
```

19.4.10 lwgpio_int_enable()

This function enables or disables GPIO interrupts for a pin on the peripheral.

Synopsis

```
void lwgpio_int_enable
(
    LWGPIO_STRUCT_PTR handle,
    boolean           ena
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by the lwgpio_init() function.
ena [in] — TRUE (enable), FALSE (disable).

Description

This function enables or disables interrupts for the specified pin (or set of pins- if so-called keyboard-interrupt peripheral is used) on the peripheral level. This effectively enables the interrupt channel from peripheral to the interrupt controller. This function does not set up interrupt controller to acknowledge interrupts. It is recommended to clear the flag with the lwgpio_int_clear_flag() function prior to the lwgpio_int_enable() function call.

Return Value

- None

Example

The following example shows how to enable the rising edge interrupt for the LWGPIO pin on MK40X256.

```
lwgpio_int_clear_flag(&btn_int);
lwgpio_int_enable(&btn_int, TRUE);
/* Enable interrupt for button on interrupt controller */
_bsp_int_init(lwgpio_get_int_vector(&btn_int), BUTTON_PRIORITY_LEVEL, 0, TRUE);
```

19.4.11 lwgpio_int_get_flag()

This function gets the pending interrupt flag on the GPIO interrupt peripheral.

Synopsis

```
boolean lwgpio_int_get_flag
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by the lwgpio_init() function.

Description

This function returns the pin interrupt flag on the peripheral. If the interrupt is so-called keyboard interrupt, it returns the interrupt flag for a set of pins.

Return Value

- TRUE if the flag is set
- FALSE if the flag is not set

Example

The following example checks the pending interrupt for the LWGPIO pin on MCF52259.

```
if (lwgpio_int_get_flag(&btn_int) == TRUE)
{
    /* do some action */
}
```

19.4.12 lwgpio_int_clear_flag()

This function clears the pending interrupt flag on the GPIO interrupt peripheral.

Synopsis

```
void lwgpio_int_clear_flag
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by the lwgpio_init() function.

Description

This function clears the pin interrupt flag on the peripheral. If the interrupt is so-called keyboard interrupt, it clears the interrupt flag for a set of pins. This is typically called from the interrupt service routine, if the peripheral requires the flag being cleared by the software.

Return Value

- None

Example

The following example clears pending interrupt for the LWGPIO pin on MCF52259.

```
lwgpio_int_clear_flag(&btn_int);
```

19.4.13 lwgpio_int_get_vector()

This function gets the interrupt vector number that belongs to the pin or a set of pins.

Synopsis

```
uint_32 lwgpio_int_get_vector
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by the lwgpio_init() function.

Description

This function returns the interrupt vector index for the specified pin. The obtained vector index can be used to install the interrupt by the MQX.

Return Value

- Vector table index to be used for installing the interrupt handler.

Example

The following example gets the vector number for the specific pin and it installs the ISR for the LWGPIO pin on MCF52259.

```
uint_32 vector = lwgpio_int_get_vector(&btn1);
_int_install_isr(vector, int_callback, (void *) param);
```

19.4.14 lwgpio_set_attribute ()

Synopsis

```
boolean lwgpio_set_attribute
(
LWGPIO_STRUCT_PTR  handle,
uint_32 attribute_id,
uint_32 value
)
```

Parameters

handle [in] - Pointer to the LWGPIO_STRUCT pre-initialized by *lwgpio_init()* function.

attribute_id [in] - Attribute identificator.

value [in] - Attribute value.

Description

MCUs have different properties for GPIO pins. These properties depend on the architecture and the GPIO or PORT module. This function handles these attributes. The attribute is defined by a special attribute ID. The value specifies requirements for the attribute (enable, disable, or a specific value). There are common attribute IDs and values placed in \io\lwgpio\lwgpio.h and driver specific attributes and values placed in \io\lwgpio\lwgpio_<driver>.h.

Return Value

- TRUE (success)
- FALSE (failure)

Example

The following example shows how to set the pull up for the button1 handle. This example returns FALSE if the pull up attribute is not available.

```
Lwgpio_set_attribute(&button1, LWGPIO_ATTR_PULL_UP, LWGPIO_AVAL_ENABLE);
```

19.5 Macro Functions Exported by the LWGPIO Driver

LWGPIO driver exports inline functions (macros) for an easy pin driving without needing to use the pin handle structure. The structure is initiated internally in the inline code. These functions are available for every platform and are generic. They are defined in the *lwgpio.h* file.

19.5.1 lwgpio_set_pin_output()

This macro puts the specified pin into the output state with the defined output value.

Synopsis

```
boolean inline lwgpio_set_pin_output(
    LWGPIO_PIN_ID id,
    LWGPIO_VALUE pin_state
)
```

Parameters

id [in] — LWGPIO_PIN_ID number identifying pin which is platform and peripheral specific.
pin_state [in] — LWGPIO_VALUE enum value for initial output control.

Description

This inline function switches the specified pin into the output state. The output level is defined by the *pin_state* parameter.

Return Value

- TRUE (success)
- FALSE (failure)

Example

The following example shows how to set high voltage level output for the LWGPIO pin PTA.3 on MCF52259.

```
lwgpio_set_pin_output(LWGPIO_PORT_TA | LWGPIO_PIN3, LWGPIO_VALUE_HIGH);
```

19.5.2 lwgpio_toggle_pin_output()

This macro changes (toggles) the output value of the specified pin and requires the pin multiplexer to be set to the GPIO function. Otherwise, the pin output is not going to change.

Synopsis

```
boolean inline lwgpio_toggle_pin_output(
    LWGPIO_PIN_ID id
)
```

Parameters

id [in] — LWGPIO_PIN_ID number identifying pin which is platform and peripheral specific.

Description

This inline function switches the specified pin into the output state and toggles the output value. The output level is taken from the output buffer value.

Return Value

- TRUE (success)
- FALSE (failure)

Example

The following example shows how to toggle output for the LWGPIO pin PTA.3 on MCF52259.

```
lwgpio_toggle_pin_output(LWGPIO_PORT_TA | LWGPIO_PIN3);
```

19.5.3 lwgpio_get_pin_input()

This function gets voltage value (low or high) of the specified pin.

Synopsis

```
LWGPIO_VALUE inline lwgpio_get_pin_input
(
    LWGPIO_STRUCT_PTR id
)
```

Parameters

id [in] — LWGPIO_PIN_ID number identifying pin which is platform and peripheral specific.

Description

This function gets the input voltage level value in the same way as lwgpio_get_value() function does.

Return Value

- LWGPIO_VALUE_HIGH - Voltage value of specified pin is high
- LWGPIO_VALUE_LOW - Voltage value of specified pin is low
- LWGPIO_VALUE_NOCHANGE - Could not configure pin for input (failure)

Example

The following example shows how to get (pre-set) voltage level for the LWGPIO pin PTA.3 on MCF52259.

```
value = lwgpio_get_pin_input(LWGPIO_PORT_TA | LWGPIO_PIN3);
if (value == LWGPIO_VALUE_NOCHANGE)
{
    printf("Can not configure pin PTA.3 for input.\n");
    _mqx_exit(-1);
}
```

19.6 Data Types Used by the LWGPIO API

The following data types are used within the LWGPIO driver.

19.6.1 LWGPIO_PIN_ID

This 32 bit number specifies the pin on the MCU. The number is MCU-specific.

```
typedef uint_32 LWGPIO_PIN_ID;
```

In general, LWGPIO_PIN_ID value consists of two logically OR-ed constants: port value and pin value. Both of these macro values have a common nomenclature across all platforms:

```
LWGPIO_PIN_ID pin_id = LWGPIO_PORT_xyz | LWGPIO_PIN_z;
```

Though these macros have common format and style, they are MCU-specific. Every MCU or platform has its own macros defined. The constants can be found in the *lwgpio_<mcu>.h* file and should be used to create the LWGPIO_PIN_ID value.

19.6.2 LWGPIO_STRUCT

A pointer to this structure is used as a handle for the LWGPIO driver API functions. The content of this structure is MCU-specific. This structure has to be allocated in the user application space such as heap and stack before calling lwgpio_init() function.

19.6.3 LWGPIO_DIR

This enumerated value specifies the direction. The value is generic.

```
typedef enum {
    LWGPIO_DIR_INPUT,
    LWGPIO_DIR_OUTPUT,
    LWGPIO_DIR_NOCHANGE
} LWGPIO_DIR;
```

The LWGPIO_DIR enum type is used to set or get the direction of the specified pin. The special value of LWGPIO_DIR_NOCHANGE can be passed to a function if the change of the direction is undesirable.

19.6.4 LWGPIO_VALUE

This enumerated value specifies the voltage value of the pin. The value is generic.

```
typedef enum {
    LWGPIO_VALUE_LOW,
    LWGPIO_VALUE_HIGH,
    LWGPIO_VALUE_NOCHANGE
} LWGPIO_VALUE;
```

The LWGPIO_VALUE enum type is used to set or get the voltage value of the specified pin. The special value of LWGPIO_VALUE_NOCHANGE can be passed to a function if the change of the value is undesirable or it is returned in special case if the value can not be obtained.

19.6.5 LWGPIO_INT_MODE

This integer value specifies the interrupt mode of the pin. The value is generic.

```
typedef uchar LWGPIO_INT_MODE;
```

In general, LWGPIO_INT_MODE value consists of several logically OR-ed constants. The same macro can have a different value on a different MCU.

```
LWGPIO_INT_MODE_RISING
LWGPIO_INT_MODE_FALLING
LWGPIO_INT_MODE_HIGH
LWGPIO_INT_MODE_LOW
```

Note that although these macros are MCU defined, it does not mean that MCU supports any combination. In case of an unsupported combination, the function with incorrect LWGPIO_INT_MODE will return the failure status.

19.7 Example

The example for the LWGPIO driver that shows how to use LWGPIO driver API functions is provided with the MQX installation and it is located in `mqx\examples\lwgpio` directory.

Chapter 20 Low Power Manager

20.1 Overview

This section describes the Low Power Manager (LPM) that accompanies the MQX RTOS. The Freescale MQX provides low power functionality in terms of run-time clock frequency changes and CPU/peripheral operation mode changes such as shutting down peripheral clocks, module enabling, and setting pin multiplexer. The feature of CPU core sleep is also available in the idle task designed for further power saving while all user tasks are blocked.

LPM is a common interface that enables the user application to switch between pre-defined low power operation modes and clock configurations in a controlled way at runtime. A user defines the behavior of the CPU core and selected low power-enabled peripheral drivers in each operation mode. Regarding the clock configurations, LPM serves as a wrapper around another MQX component, the Clock Manager.

The purpose of LPM is to gather all information needed for low power system change and manage the preparation and recovery phases with a few function calls. Drivers, stacks, and other user state machines which are affected by different low power settings register their callback handlers at the LPM. These handlers are used by the LPM to notify all registered drivers before any operation mode change and before and after the clock configuration change. The drivers have to adapt to the new global settings within the callbacks according to their behavior specified for the operation mode and clock configuration. All registered drivers are accessed by the LPM in a user-defined order called a "dependency level".

LPM implements the custom C language API and does not follow the standard POSIX-like driver interface (I/O Subsystem).

LPM functionality is currently available for Kinetis platforms only and must be explicitly enabled in *user_config.h* using the MQX_ENABLE_LOW_POWER configuration option.

The system timer and serial driver are currently the only low power-enabled drivers. For more information about low power mode implementation in a particular driver, see “Low Power Support” chapter in the corresponding driver section.

20.2 Source Code Location

LPM module is a part of the BSP library since the behavior is related to a particular board and to peripheral drivers. There are low power definitions and behavior structures defined for the CPU core *(init_lpm.h, init_lpm.c)* and for supported drivers *(init_sci.c)* in the BSP directory.

The source code files for the LPM are located in source\io\lpm directory. The *lpm_* file prefix is used for all LPM module related API files. Additional functionality is also added to the source code for all low power-enabled drivers.

20.3 Header Files

To use the LPM functionality, include the *bsp.h* header file into your application. It already contains all needed header file and includes all definitions.

20.4 API Function Reference

This section serves as a function reference for the LPM module.

20.4.1 _lpm_install()

The function installs and enables the Low Power Manager within the MQX.

Synopsis

```
_mqx_uint _lpm_install
(
    const LPM_CPU_OPERATION_MODE _PTR_ operation_modes,
    LPM_OPERATION_MODE                 default_mode
)
```

Parameters

operation_modes [in] - Pointer of the CPU core operation modes array.

default_mode [in] - Enumerated value of default (current) operation mode.

Description

This function installs the MQX LPM with given CPU core behavior specification in all operation modes and with default (currently running) operation mode. Driver registrations and power mode switching is possible after the successful return from this function.

By default, this function is called in the BSP startup code before any driver installation takes place, so the user application shouldn't call this function again.

Return Value

- MQX_INVALID_PARAMETER - Wrong parameter
- MQX_COMPONENT_EXISTS - LPM already installed
- MQX_IO_OPERATION_NOT_AVAILABLE - Possible memory problem
- MQX_OK - Success

Example

The following example shows the installation of the LPM:

```
if (MQX_OK != _lpm_install (LPM_CPU_OPERATION_MODES, LPM_OPERATION_MODE_RUN))
{
    printf ("Error during LPM install!\n");
}
```

20.4.2 _lpm_uninstall()

The function uninstalls LPM functionality from the MQX.

Synopsis

```
_mqx_uint _lpm_uninstall
(
    void
)
```

Parameters

None.

Description

This function uninstalls the LPM from the MQX leaving current operation mode and clock configuration unchanged. No more LPM function calls may occur in the application after this function is called.

Return Value

- MQX_IO_OPERATION_NOT_AVAILABLE - LPM not installed or memory problem
- MQX_OK - Success

20.4.3 _lpm_register_driver()

This function registers a driver that must be notified about low power system changes at LPM.

Synopsis

```
_mqx_uint _lpm_register_driver
(
    const LPM_REGISTRATION_STRUCT_PTR driver_registration_ptr,
    const pointer                     driver_specific_data_ptr,
    _mqx_uint_ptr                     registration_handle_ptr
)
```

Parameters

driver_registration_ptr [in] - Pointer to a registration structure with driver callbacks.

driver_specific_data_ptr [in] - Pointer to a driver specific data to be passed to callbacks.

registration_handle_ptr [out] - Unique driver registration handle.

Description

This function registers notification callbacks of the driver for operation mode changes and for clock configuration changes. For operation mode changes, the corresponding callback is called before the actual mode change. For clock configuration changes, the corresponding callback is called both before and after the change is made. The callback routines have unified API as described below.

The callback handler can return an error which means that the driver is not ready or able to switch to a given low power mode. In this case, no low power system change is made and a rollback takes place. The rollback is done by notifying all drivers already processed (except for the one whose callback caused an error) in reverse order and with the original mode specified.

Besides callbacks, there is also a dependency level specified for each driver during registration. The dependency level affects the order in which the registered drivers are notified about low power system changes. For pre-change notifications, the lower dependency level drivers are processed first. The order of registration is used for drivers at the same dependency level. For post- notifications or for rollback notifications in case of failure, the order is reversed.

If the driver registration succeeds, the function returns MQX_OK and unique driver registration handle that must be used later during the driver unregister process.

Low power-enabled POSIX drivers in MQX register themselves at the LPM automatically during their installation, so user application shouldn't register them explicitly again.

Return Value

- MQX_INVALID_PARAMETER - Wrong parameter
- MQX_IO_OPERATION_NOT_AVAILABLE - LPM not installed
- MQX_OUT_OF_MEMORY - Possible memory problem
- MQX_OK - Success

Example

The following example shows the automatic registration of polled serial driver into LPM:

```
IO_SERIAL_POLLED_DEVICE_STRUCT_PTR  dev_ptr;
LPM_REGISTRATION_STRUCT             registration;

registration.CLOCK_CONFIGURATION_CALLBACK =
    _io_serial_polled_clock_configuration_callback;

registration.OPERATION_MODE_CALLBACK      =
    _io_serial_polled_operation_mode_callback;

registration.DEPENDENCY_LEVEL             =
    BSP_LPM_DEPENDENCY_LEVEL_SERIAL_POLLED;

result = _lpm_register_driver
            (
                &registration, dev_ptr,
                &(dev_ptr->LPM_INFO.REGISTRATION_HANDLE)
            );

if (MQX_OK == result)
{
    _lwsem_create (&(dev_ptr->LPM_INFO.LOCK), 1);
    dev_ptr->LPM_INFO.FLAGS = 0;
}
```

20.4.4 _lpm_unregister_driver()

The function unregisters driver from the LPM.

Synopsis

```
_mqx_uint _lpm_unregister_driver
(
    _mqx_uint registration_handle
)
```

Parameters

registration_handle [in] - Unique driver registration handle.

Description

This function unregisters a driver from the LPM by using the unique handle returned by registration function. If the function succeeds, the driver keeps current low power settings and it's not notified anymore about low power system changes.

Return Value

- MQX_INVALID_PARAMETER - Wrong parameter
- MQX_IO_OPERATION_NOT_AVAILABLE - LPM not installed
- MQX_INVALID_HANDLE - Corresponding driver not registered at LPM
- MQX_OK - Success

Example

The following example shows the automatic unregister process of polled serial driver from LPM:

```
IO_SERIAL_POLLED_DEVICE_STRUCT_PTR dev_ptr;

_lpm_unregister_driver (dev_ptr->LPM_INFO.REGISTRATION_HANDLE);
_lwsem_destroy (&(dev_ptr->LPM_INFO.LOCK));
```

20.4.5 _lpm_set_clock_configuration()

The function switches the system to a given low power clock configuration including all preparation and recovery actions.

Synopsis

```
_mqx_uint _lpm_set_clock_configuration
(
    BSP_CLOCK_CONFIGURATION clock_configuration
)
```

Parameters

clock_configuration [in] - Clock configuration identifier defined in the BSP.

Description

This function notifies all registered drivers about the clock configuration that is to be switched to. The pre-notifications are made in ascending order according to the driver dependency level or according to the order of registration in case of the same dependency level. After pre-notifications, this function changes the clock configuration physically and post-notifies all drivers in reverse order that the clock configuration has been changed.

In case of any failure or error reported by any driver during pre-notifications phase, the clock configuration is not changed and all drivers already processed are notified again in reverse order with the original clock information.

Return Value

- MQX_INVALID_PARAMETER - Wrong parameter
- MQX_IO_OPERATION_NOT_AVAILABLE - LPM not installed or clock configuration change failed
- MQX_OK - Success

Example

The following example shows the clock configuration change:

```
if (MQX_OK != _lpm_set_clock_configuration(BSP_CLOCK_CONFIGURATION_1))
{
        printf ("Clock configuration not changed!\n");
}
```

20.4.6 _lpm_get_clock_configuration()

The function returns the identifier for the clock configuration in which the MQX is currently running.

Synopsis

```
BSP_CLOCK_CONFIGURATION _lpm_get_clock_configuration
(
    void
)
```

Parameters

None

Description

This function returns active clock configuration or -1 if the LPM is not installed.

Return Value

- BSP_CLOCK_CONFIGURATION - One of the predefined enumerated values
- -1 - When LPM is not installed

Example

The following example shows acquiring the current clock configuration:

```
clock_configuration = _lpm_get_clock_configuration();
```

20.4.7 _lpm_set_operation_mode()

The function switches the system to given low power operation mode and performs all preparation and recovery actions.

Synopsis

```
_mqx_uint _lpm_set_operation_mode
(
    LPM_OPERATION_MODE operation_mode
)
```

Parameters

operation_mode [in] - Operation mode identifier defined in the BSP.

Description

This function pre-notifies all registered drivers about the operation mode that is to be switched to. The notifications are made in ascending order according to the driver dependency level or according to the order of registration in case of the same dependency level. After the notifications, this function actually changes the power mode of the CPU core.

In case of any failure or error reported by any driver during the pre-notifications phase, the power mode of the CPU core is not changed and all already processed drivers are notified in reverse order with the original operation mode parameter.

NOTE

> This function may block the CPU core and may not return until specified wakeup event occurs. It may also restart the idle task to enable/disable the idle task sleep feature.

Return Value

- MQX_INVALID_PARAMETER - Wrong parameter
- MQX_INVALID_CONFIGURATION - Wrong CPU core operation mode settings
- MQX_IO_OPERATION_NOT_AVAILABLE - LPM not installed or operation mode change failed
- MQX_OK - Success

Example

The following example shows the operation mode change:

```
if (MQX_OK != _lpm_set_operation_mode(LPM_OPERATION_MODE_WAIT))
{
    printf ("Operation mode not changed!\n");
}
```

20.4.8 _lpm_get_operation_mode ()

The function returns identifier of the operation mode in which the MQX is currently running.

Synopsis

```
LPM_OPERATION_MODE _lpm_get_operation_mode
(
    void
)
```

Parameters

None.

Description

This function returns active operation mode or -1 if the LPM is not installed.

Return Value

- LPM_OPERATION_MODE - One of the predefined enumerated values
- -1 - When LPM is not installed

Example

The following example shows acquiring the current operation mode:

```
operation_mode = _lpm_get_operation_mode();
```

20.4.9 _lpm_wakeup_core()

This platform-specific function signals the CPU core not to return to sleep mode again after the ISR finishes. This function should be called from the ISR.

Synopsis

```
void _lpm_wakeup_core
(
    void
)
```

Parameters

None.

Description

One of the possible low power operation modes is "execute interrupts only", so the CPU core has no chance to exit this mode without cooperation from the interrupt service routine. To pass control back to the tasks, the application must use this function within an interrupt routine to keep the CPU running after the ISR finishes.

This function is currently available for Kinetis platform only. It clears the SLEEPONEXIT flag in the Kinetis system control register.

Return Value

None.

Example

The following example shows how to keep the CPU core awake after ISR:

```
void ISR (pointer data)
{
...
    if (time_to_let_the_tasks_run)
    {
        _lpm_wakeup_core();
    }
}
```

20.4.10 _lpm_idle_sleep_setup()

This function enables/disables the feature of the CPU core sleep in idle task. It may also restart the idle task.

Synopsis

```
void _lpm_idle_sleep_setup
(
    boolean enable
)
```

Parameters

enable [in] - Enable or disable the feature.

Description

The function enables or disables the feature CPU core sleep during the execution of the idle task. For that purpose, the function may restart the idle task. When enabled, the idle task sleep feature is relevant only in the CPU power modes RUN, WAIT, VLPR, and VLPW.

Return Value

None.

Example

The following example shows how to enable idle task sleep feature:

```
_lpm_idle_sleep_setup(TRUE);
```

20.4.11 _lpm_idle_sleep_check()

This function checks whether it's possible to perform idle task sleep in the current operation mode.

Synopsis

```
boolean _lpm_idle_sleep_check
(
```

```
    void
)
```

Parameters

None.

Description

The function checks current settings of the power mode registers to find out whether it is possible to put CPU core to sleep in the idle task. This function is used internally by the LPM and is available for information purposes only.

Return Value

- TRUE - Current settings allow to execute core sleep in idle task.
- FALSE - Otherwise.

Example

The following example shows how to check idle sleep feature availability in the current operation mode:

```
sleep_allowed = _lpm_idle_sleep_check();
```

20.4.12 driver_notification_callback()

All driver notification functions must be defined with the following unified type.

Synopsis

```
LPM_NOTIFICATION_RESULT driver_notification_callback
(
    LPM_NOTIFICATION_STRUCT_PTR notification_ptr,
    pointer                     driver_specific_data_ptr
)
```

Parameters

notification_ptr [in] - Notification type, target operation mode, and clock configuration.

driver_specific_data_ptr [in] - Pointer to the driver specific data that was passed to the LPM during driver registration.

Description

Notification callback should change HW settings of the driver according to the given low power identifiers and driver specific behavior structures that are part of its initialization information. Notification callback can return an error to indicate that the low power change cannot be fulfilled. This causes a rollback during a particular operation mode change or clock configuration change.

There can be a significant time delay between pre-notification and post-notification. Therefore, locking the access to the driver in the meantime may be necessary to avoid any unexpected behavior.

Return Value

- LPM_NOTIFICATION_RESULT_OK - Success
- LPM_NOTIFICATION_RESULT_ERROR - Driver not ready or able to switch to given low power settings

Example

The following example shows possible serial clock configuration callback:

```
LPM_NOTIFICATION_RESULT serial_clock_configuration_callback
(
    /* [IN] Low power notification */
    LPM_NOTIFICATION_STRUCT_PTR notification_ptr,
    /* [IN/OUT] Device specific data */
    pointer                     driver_specific_data_ptr
)
{
    if (LPM_NOTIFICATION_TYPE_PRE == notification->NOTIFICATION_TYPE)
    {
        if (   (BSP_CLOCK_CONFIGURATION_2MHZ == notification->CLOCK_CONFIGURATION)
            && (LPM_OPERATION_MODE_RUN == notification->OPERATION_MODE))
        {
            /* Unable to operate when on 2MHZ */
            return LPM_NOTIFICATION_RESULT_ERROR;
        }
    }

    if (LPM_NOTIFICATION_TYPE_POST == notification->NOTIFICATION_TYPE)
    {
        SERIAL_INIT_STRUCT_PTR  init;
        uint_32                 input_clock;

        init = ((SERIAL_STRUCT_PTR)driver_specific_data_ptr)->DEV_INIT_DATA_PTR;

        input_clock = cm_get_clock
                      (
                          notification->CLOCK_CONFIGURATION,
                          init->CLOCK_SOURCE
                      );

        serial_set_baudrate
                      (
                          driver_specific_data_ptr,
                          input_clock,
                          init->BAUD_RATE
                      );
    }

    return LPM_NOTIFICATION_RESULT_OK;
}
```

20.5 Remarks

The points to consider when working with LPM in the MQX:

- All functions described above are thread-safe and should not be called from an ISR, except in the explicitly stated cases. These functions should also not be called from the notification callbacks of registered drivers. In particular, the idle task sleep feature cannot be switched from within any ISR.
- The LPM is automatically installed during BSP initialization before any other driver installation, so it should not be done again by the application.
- All low power-enabled POSIX drivers register/unregister themselves automatically during their installation/uninstallation, so it shouldn't be done again by the application.
- The Kinetis VLLSx and BAT modes are not currently supported. In the *init_lpm.c* file, mapping exists between generic operation mode identifiers and the supported Kinetis CPU core power modes. This mapping can be changed by the user. All available Kinetis operation modes can be found in the *lpm_kinetis.h* file.
- After wakeup from an operation mode where CPU is inactive, the system remains in the last operation mode set. It is an application's responsibility to switch to another mode immediately after wakeup by calling *_lpm_set_operation_mode()* function.
- Idle sleep feature may cause problems while debugging. It is recommended to turn the feature off for debug purposes. The feature is disabled by default and as long as the LPM is not installed.

20.6 Data Types Used by the LPM Driver API

The following data types are defined regarding the LPM functionality.

20.6.1 LPM_OPERATION_MODE

This enumerated type defines identifiers of the generic operation modes available in the BSP and their overall count. Arrays of structures are defined which describe the behavior of the CPU core and each low power-enabled peripheral for all of the following operation modes in the BSP.

```
typedef enum
{
    LPM_OPERATION_MODE_RUN = 0,
    LPM_OPERATION_MODE_WAIT,
    LPM_OPERATION_MODE_SLEEP,
    LPM_OPERATION_MODE_STOP,
    LPM_OPERATION_MODES
} LPM_OPERATION_MODE;
```

20.6.2 LPM_NOTIFICATION_TYPE

This enumerated type specifies whether the driver notification is done before or after the actual low power system change. It is passed to all notification callbacks.

```
typedef enum {
    LPM_NOTIFICATION_TYPE_PRE,
    LPM_NOTIFICATION_TYPE_POST
} LPM_NOTIFICATION_TYPE;
```

20.6.3 LPM_NOTIFICATION_RESULT

One of these enumerated values should be returned by any notification callback. When LPM_NOTIFICATION_RESULT_ERROR is returned, it forces LPM not to make low power system changes and to rollback all already processed drivers to the previous mode.

```
typedef enum {
    LPM_NOTIFICATION_RESULT_OK,
    LPM_NOTIFICATION_RESULT_ERROR
} LPM_NOTIFICATION_RESULT;
```

20.6.4 LPM_NOTIFICATION_STRUCT

A pointer to this structure is passed to all notification callback handlers to inform them about a type of notification and about low power settings to be switched to.

```
typedef struct lpm_notification_struct {
    LPM_NOTIFICATION_TYPE   NOTIFICATION_TYPE;
    LPM_OPERATION_MODE      OPERATION_MODE;
    BSP_CLOCK_CONFIGURATION CLOCK_CONFIGURATION;
} LPM_NOTIFICATION_STRUCT, _PTR_ LPM_NOTIFICATION_STRUCT_PTR;
```

20.6.5 LPM_REGISTRATION_STRUCT

This structure has to be filled and passed during the driver registration at the LPM. It specifies both the operation mode and clock configuration callbacks and the order of processing among other registered drivers. One of the callbacks can also be NULL if not required. See example below:

```
typedef struct lpm_registration_struct {
    LPM_NOTIFICATION_CALLBACK CLOCK_CONFIGURATION_CALLBACK;
    LPM_NOTIFICATION_CALLBACK OPERATION_MODE_CALLBACK;
    _mqx_uint                 DEPENDENCY_LEVEL;
} LPM_REGISTRATION_STRUCT, _PTR_ LPM_REGISTRATION_STRUCT_PTR;
```

20.7 Platform-Specific Data Types Used by the LPM API

The following data types are used in generic LPM API calls, but are defined differently on each processor platform. So far only the Kinetis platform is supported by the MQX LPM.

20.7.1 LPM_CPU_POWER_MODE_INDEX

The enumerated type defines identifiers for all supported Kinetis specific CPU core power modes and their overall count.

```
typedef enum
{
    LPM_CPU_POWER_MODE_RUN = 0,
    LPM_CPU_POWER_MODE_WAIT,
    LPM_CPU_POWER_MODE_STOP,
    LPM_CPU_POWER_MODE_VLPR,
    LPM_CPU_POWER_MODE_VLPW,
    LPM_CPU_POWER_MODE_VLPS,
    LPM_CPU_POWER_MODE_LLS,
    LPM_CPU_POWER_MODES
} LPM_CPU_POWER_MODE_INDEX;
```

20.7.2 LPM_CPU_OPERATION_MODE

The platform-specific structure describes the behavior of the CPU core in one of the operation modes available. It maps from one of the generic operation modes to one of the Kinetis-specific CPU core power modes. Additional operation mode flag can be specified here. Also wakeup settings of LLWU registers can be specified here (applies only for Kinetis LLS mode).

```
typedef struct lpm_cpu_operation_mode {
    LPM_CPU_POWER_MODE_INDEX MODE_INDEX;
    uint_8                      FLAGS;
    uint_8                      PE1;
    uint_8                      PE2;
    uint_8                      PE3;
    uint_8                      PE4;
    uint_8                      ME;
} LPM_CPU_OPERATION_MODE, _PTR_ LPM_CPU_OPERATION_MODE_PTR;
```

Table 20-1. LPM_CPU_OPERATION_MODE Flags

FLAGS	Description
LPM_CPU_POWER_MODE_FLAG_SLEEP_ON_EXIT	This flag tells the LPM that specified operation mode is "execute interrupts only", i.e. the CPU core is active only for the interrupt service routines and it returns to sleep upon exit out of any ISR. Applies only for Kinetis wait and stop modes. To pass the control back to tasks, function *_lpm_wakeup_core()* must be called within any ISR.

20.8 Example

The example of the low power feature that shows how to use LPM API functions is provided together with the MQX installation and is located in mqx\examples\lowpower directory.

The default settings of operation modes, clock configurations, and behavior definitions for the CPU core and low power enabled drivers can be found in the corresponding BSP directory.

Chapter 21 Resistive Touch Screen Driver

21.1 Overview

This chapter describes the device driver which is a common interface for four-wire resistive touch screens as shown on Figure 21-1.

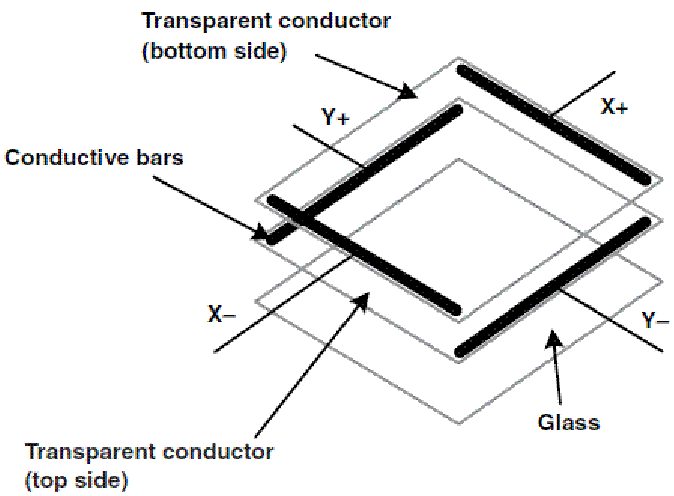


Figure 21-1. Four-wire Resistive Touch Screen

The touch screen driver uses Lightweight GPIO driver to toggle voltage on depicted electrodes and MQX ADC driver for measuring voltage on electrodes marked as X+ and Y+.

The x and y coordinates of a touch are read in two steps as described below:

- Before the measurement all electrodes are grounded, set to low using LWGPIO driver, to discharge electrodes.
- Measuring of X coordinate - Before the measurement X+ is driven to high (Ucc), the X- grounded and Y- set to high impedance using LWGPIO driver. The position is measured on Y+ electrode using the ADC driver.
- When a touch is detected, the electrodes are grounded again and the measurement continues analogically to measure Y coordinate.

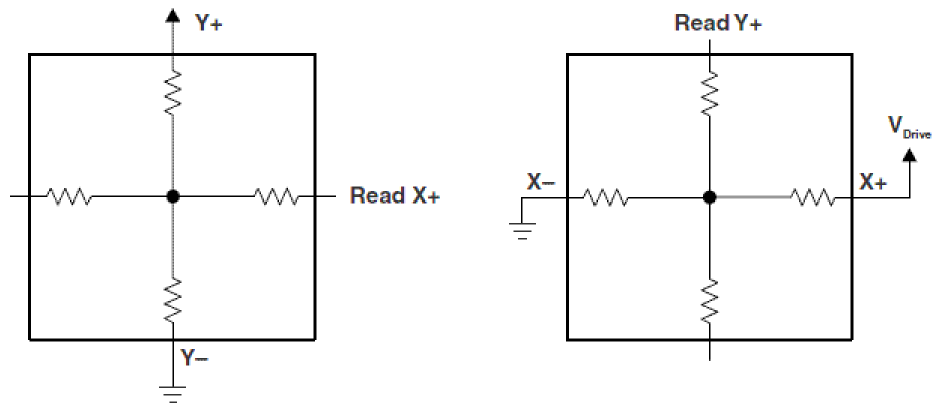


Figure 21-2. Measuring of Voltage on Electrodes X+ and Y+

For touch screen functionality, it is crucial to wire electrodes with MCU. The X+ and Y+ pins must offer both GPIO and ADC functionality for both. It is also very important for measuring that the correct settings of ADC limits are used for touch detection.

21.2 Source Code Location

Driver	Location
TCHRES generic driver	`source\io\tchres`
TCHRES hardware-specific driver	`source\bsp\<board>\init_tchres.c`

21.3 Header Files

To use the TCHRES device driver, include the header file from *source\io\tchres* in your application or in the BSP file *bsp.h*. Use the header file according to the following table.

Driver	Header File
TCHRES driver	`tchres.h`

The file *tchres_prv.h* contains private constants and data structures that TCHRES device driver uses.

21.4 Installing Drivers

TCHRES device driver provides installation function *_io_tchres_install()* called by the user application. The driver installation routine calls *_io_dev_install_ext()* internally.

Example of *_io_dev_install* function call:

```
_io_tchres_install("tchscr:", &_bsp_tchscr_resisitve_init, &install_params);
```

The *_bsp_tchscr_resisitve_init* is the initialization structure containing information for TCHRES driver. The install_params points to the installation parameters structure containing information about ADC devices to be used for measuring.

Initialization structure contains, among other values, also the TCHRES_ADC_LIMITS_STRUCT which is important for measuring.

```
/* Adc limits struct */
typedef struct tchres_adc_limits
{
    uint_16   FULL_SCALE;
    int_16    X_TOUCH_MIN;
    int_16    Y_TOUCH_MIN;
    int_16    X_TOUCH_MAX;
    int_16    Y_TOUCH_MAX;
} TCHRES_ADC_LIMITS_STRUCT, _PTR_ TCHRES_ADC_LIMITS_STRUCT_PTR;
```

A full scale should reflect ADC resolution. For example, for 12-bit ADC it should be set to 0xFFF. Minimum and maximum for x and y coordinate is used for filtering samples measured on X+ and Y+ electrodes. Samples out of this range will be interpreted as no touch.

```
/* install parameters - adc devices used for measuring on X+ and Y+ electrodes */
typedef struct tchres_install_param_struct
{
char_ptr ADC_XPLUS_DEVICE; /* ADC device for X+ electrode */
char_ptr ADC_YPLUS_DEVICE; /* ADC device for Y+ electrode */
} TCHRES_INSTALL_PARAM_STRUCT, _PTR_ TCHRES_INSTALL_PARAM_STRUCT_PTR;
```

Installation parameters are used to provide string identifiers of ADC devices used for measuring on X+ and Y+ electrodes. Installation parameters should be provided by the user application which is also responsible for their opening prior to calling the driver installation routine.

21.5 Driver Services

TCHRES driver provides following services:

API	Calls
_io_fopen()	_tchres_open()
_io_fclose()	_tchres_close()
_io_ioctl()	_tchres_ioctl()

21.5.1 Opening TCHRES Device

Prior to using the touch screen device, it must be installed and opened. Since there is no need for any further work, the second parameter should be set to null as show in an example below.

```
FILE_PTR tchscr_dev = fopen("tchscr:", NULL);
```

Since there is no read or write function defined for the device, the communication is handled only by I/O control commands.

21.6 I/O Control Commands

This section describes I/O control commands used in *_io_ioctl()* calls on TCHRES device which are defined in *tchres.h*.

Command	Description	Parameters
IO_IOCTL_TCHSCR_GET_POSITION_RAW	Command measures touch position which is returned back in raw ADC values. Return code is either one of error code prefixed by TCHRES_ERROR_ or TCHRES_OK when touch was detected. Return codes are defined in tchres.h.	*param_ptr* - pointer to TCHRES_POSITION_STRUCT used for passing back touch result which is valid only on TCHRES_OK.
IO_IOCTL_TCHSCR_GET_RAW_LIMITS	Command returns ADC limits used for touch detection through parameter passed directly to ioctl as an argument. Return code is MQX_OK or TCHRES_ERROR_INVALID_PARAMETER.	*param_ptr* - pointer to TCHRES_ADC_LIMITS_STRUCT

21.7 Data Types

21.7.1 TCHRES_INIT_STRUCT

Synopsis:

```
typedef struct tchres_init_struct {
   TCHRES_PIN_CONFIG_STRUCT PIN_CONFIG;
   TCHRES_ADC_LIMITS_STRUCT ADC_LIMITS;
   LWGPIO_PIN_ID ADC_CHANNEL_X_SOURCE;
   ADT_TRIGGER_MASK ADC_CHANNEL_X_TRIGGER;
   LWGPIO_PIN_ID ADC_CHANNEL_Y_SOURCE;
   ADT_TRIGGER_MASK ADC_CHANNEL_Y_TRIGGER;
} TCHRES_INIT_STRUCT, _PTR_ TCHRES_INIT_STRUCT_PTR;
```

Parameters:

PIN_CONFIG - Pins connected to touch screen electrodes.

ADC_LIMITS - Limits for ADC used for touch detection.

ADC_CHANNEL_X_SOURCE - ADC channel for X+ electrode.

ADC_CHANNEL_X_TRIGGER - Trigger mask for X+ ADC channel.

ADC_CHANNEL_Y_SOURCE - ADC channel for Y+ electrode.

ADC_CHANNEL_Y_TRIGGER - Trigger mask for Y+ ADC channel.

21.7.2 TCHRES_PIN_CONFIG_STRUCT

Synopsis:

```
typedef struct tchres_pin_config_struct {
   LWGPIO_PIN_ID X_PLUS;
   LWGPIO_PIN_ID X_MINUS;
   LWGPIO_PIN_ID Y_PLUS;
   LWGPIO_PIN_ID Y_MINUS;
   TCHRES_PIN_FUNCT_STRUCT PIN_FUNCT;
} TCHRES_PIN_CONFIG_STRUCT, _PTR_ TCHRES_PIN_CONFIG_STRUCT_PTR;
```

Parameters:

X_PLUS - X+ electrode GPIO pin definition.

X_MINUS - X- electrode GPIO pin definition.

Y_PLUS - Y+ electrode GPIO pin definition.

Y_MINUS - Y- electrode GPIO pin definition.

PIN_FUNCT - GPIO and ADC pin multiplexer masks.

21.7.3 TCHRES_PIN_FUNCT_STRUCT

Synopsis:

```
typedef struct tchres_pin_funct_struct {
   uint_32 X_PLUS_GPIO_FUNCTION;
   uint_32 X_PLUS_ADC_FUNCTION;
   uint_32 Y_PLUS_GPIO_FUNCTION;
   uint_32 Y_PLUS_ADC_FUNCTION;
   uint_32 X_MINUS_GPIO_FUNCTION;
   uint_32 Y_MINUS_GPIO_FUNCTION;
} TCHRES_PIN_FUNCT_STRUCT, _PTR_ TCHRES_PIN_FUNCT_STRUCT_PTR;
```

Parameters:

X_PLUS_GPIO_FUNCTION - X+ electrode GPIO pin mux mask.

X_PLUS_ADC_FUNCTION - X+ electrode ADC pin mux mask.

Y_PLUS_GPIO_FUNCTION - Y+ electrode GPIO pin mux mask.

Y_PLUS_ADC_FUNCTION - Y+ electrode ADC pin mux mask.

X_MINUS_GPIO_FUNCTION - X- electrode GPIO pin mux mask.

Y_MINUS_GPIO_FUNCTION - Y- electrode GPIO pin mux mask.

21.7.4 TCHRES_ADC_LIMITS_STRUCT

Synopsis:

```
typedef struct tchres_adc_limits {
   uint_16 FULL_SCALE;
   int_16 X_TOUCH_MIN;
   int_16 Y_TOUCH_MIN;
   int_16 X_TOUCH_MAX;
   int_16 Y_TOUCH_MAX;
} TCHRES_ADC_LIMITS_STRUCT, _PTR_ TCHRES_ADC_LIMITS_STRUCT_PTR;
```

Parameters:

FULL_SCALE - ADC resolution dependent parameter.

X_TOUCH_MIN - Min value for x-coordinate touch detection.

Y_TOUCH_MIN - Min value for y-coordinate touch detection.

X_TOUCH_MAX - Max value for x-coordinate touch detection.

Y_TOUCH_MAX - Max value for y-coordinate touch detection.

21.7.5 TCHRES_POSITION_STRUCT

Synopsis:

```
typedef struct tchres_position {
   int_16 X;
   int_16 Y;
} TCHRES_POSITION_STRUCT, _PTR_ TCHRES_POSITION_STRUCT_PTR;
```

Parameters:

X - Touch position x-coordinate.

Y - Touch position y-coordinate.

21.8 Example

For basic use, see the MQX examples. The touch screen example is located in the directory mqx\examples\tchres. TCHRES demo application is written for tower system with connected TWR-LCD board.

TCHRES device typical usage is as follows:

1. TCHRES device installation requires ADC device(s) to be opened:

```
adc_file = fopen(BSP_TCHRES_ADC_DEVICE, (const char*)&adc_init);
```

2. Preparing install parameters (one ADC device for both X+ and Y+):

```
install_params.ADC_XPLUS_DEVICE = install_params.ADC_YPLUS_DEVICE =
BSP_TCHRES_ADC_DEVICE;
```

3. When ADC device is successfully opened TCHRES can be installed:

```
_io_tchres_install("tchscr:", &_bsp_tchscr_resisitve_init, &install_params);
```

4. Before reading from TCHRES device it has to be opened:

```
tchscr_dev = fopen("tchscr:", NULL);
```

5. Read touch position using IOCTL:

```
if (_io_ioctl(tchscr_dev, IO_IOCTL_TCHSCR_GET_POSITION_RAW, &position) ==
TCHRES_OK) {
    printf("Touch detected (%d, %d)\n", position.X, position.Y);
}
```

21.9 Error Codes

Error code	Description
TCHRES_ERROR_INVALID_PARAMETER	Given parameter is invalid or NULL.
TCHRES_ERROR_NO_TOUCH	No touch detected, measured value is out of ADC limits range.
TCHRES_ERROR_TIMEOUT	When waiting for the screen surface preparation reach timeout.

Chapter 22 LWADC Driver

22.1 Overview

This section describes the Light-Weight ADC (LWADC) driver that accompanies the MQX. This driver is a common interface for ADC modules.

LWADC driver implements custom API and does not follow the standard driver interface (I/O Subsystem). Therefore, it can be used before the I/O subsystem of the MQX is initialized.

22.2 Source Code Location

The source files for the LWADC driver are located in `\mqx\source\io\lwadc` directory. *_lwadc* file prefix is used for all LWADC driver related files.

22.3 Header Files

To use LWADC driver, include the *lwadc.h* header file and the platform specific header file (e.g. *lwadc_mpxs30.h*) in your application or in the BSP header file (*bsp.h*). The platform specific header should be included before *lwadc.h*.

22.4 API Function Reference

This section contains the function reference for the LWADC driver.

22.4.1 _lwadc_init()

Synopsis

```
uint_32 _lwadc_init
(
  const LWADC_INIT_STRUCT * init_ptr
)
```

Return Value

- TRUE (Success)
- FALSE (Failure)

Parameters

init_ptr [in] — Pointer to the device specific initialization information such as ADC device number, frequency, etc.

Description

This function initializes the ADC module according to the parameters given in the platform specific initialization structure. Call to this function does not start any ADC conversion. This function is normally called in the BSP initialization code. The initialization structures for particular devices are described in a separate subsection below.

22.4.2 _lwadc_init_input()

Synopsis

```
boolean lwadc_init_input(
  LWADC_STRUCT_PTR lwadc_ptr,
  uint_32          input
)
```

Parameters

lwadc_ptr [out] — Pointer to the application allocated context structure identifying the input.
input [in] — Input specification containing ADC device and MUX input.

Return Value

- TRUE (Success)
- FALSE (Failure)

Description

This function initializes the application allocated LWADC_STRUCT (which is device-specific) with all data needed later for quick control of particular input. The structure initialized here is used in all subsequent calls to other LWADC driver functions and uniquely identifies the input. To identify ADC input, platform specific input ID number is used. The function sets the ADC input to continuous conversion mode if not already in this mode.

22.4.3 _lwadc_read_raw()

Synopsis

```
boolean _lwadc_read_raw
(
  LWADC_STRUCT_PTR lwadc_ptr,
  LWADC_VALUE *    outValue
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.
outValue [out] — Pointer to location to store read result.

Return Value

- TRUE (Success)
- FALSE (Failure)

Description

Read the current value of the ADC input and return the result without applying any scaling.

22.4.4 _lwadc_read()

Synopsis

```
boolean _lwadc_read
(
  LWADC_STRUCT_PTR lwadc_ptr,
  LWADC_VALUE    * outValue
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.
outValue [out] — Pointer to a location to store the read result.

Return Value

- TRUE (success)
- FALSE (failure)

Description

Reads the current value of the ADC input, applies scaling according to preset parameters, see _lwadc_set_attribute() function below, and returns the result.

22.4.5 _lwadc_read_average()

Synopsis

```
boolean _lwadc_read_average
(
  LWADC_STRUCT_PTR lwadc_ptr,
  uint_32          num_samples,
  LWADC_VALUE    * outValue
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.
num_samples [in] — Number of samples to read.
outValue [out] — Pointer to location to store read result.

Return Value

- TRUE (success)
- FALSE (failure)

Description

Reads num_sample samples from the specified input and returns the scaled average reading.

22.4.6 _lwadc_set_attribute()

Synopsis

```
boolean lwadc_set_attribute
(
  LWADC_STRUCT_PTR lwadc_ptr,
  LWADC_ATTRIBUTE  attribute,
  uint_32          value
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.
attribute_id [in] — Attribute to enable/disable on the specified input.
value [out] — Value for the attribute.

Return Value

- TRUE (Success)
- FALSE (Failure)

Description

This function sets attributes for the specified ADC input. Attributes could include single/differential mode, reference, scaling numerator or denominator, etc. The following table summarizes all attributes.

ATTRIBUTE	Used to set or obtain:
LWADC_RESOLUTION	ADC Device resolution in steps.
LWADC_REFERENCE	ADC Reference voltage in millivolts.
LWADC_ FREQUENCY	ADC module base frequency, actual relation between this parameter and sampling rate parameter is device specific.
LWADC_DIVIDER	The input divider.
LWADC_DIFFERENTIAL	Enables channel as a differential input.
LWADC_POWER_DOWN	Power up or down the ADC Device.
LWADC_NUMERATOR	Numerator to be used on this channel for channel scaling.
LWADC_DENOMINATOR	Denominator to be used on this channel for channel scaling.
LWADC_FORMAT	Channel data format (such as left/right aligned).
LWADC_INPUT_CONVERSION_ENABLE	Enable or disable conversion for the input.

NOTE

Not all ADC devices will support all attributes, nor will all ADCs support a per-input setting of the attributes. Setting an attribute on one input may affect other or all inputs on a device.

22.4.7 _lwadc_get_attribute()

Synopsis

```
boolean _lwadc_get_attribute
(
  LWADC_STRUCT_PTR lwadc_ptr,
  LWADC_ATTRIBUTE  attribute,
  uint_32_ptr      value
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.
attribute_id [in] — Attribute to obtain on the specified input.
value [out] — Pointer to the value for the attribute.

Return Value

- TRUE (Success)
- FALSE (Failure)

Description

This function gets attributes for the specified ADC input or for the ADC module as a whole. Attributes could include single/differential mode, reference, scaling numerator or denominator, etc. See also _lwadc_set_attribute().

22.4.8 _lwadc_wait_next()

Synopsis

```
boolean lwadc_wait_next
(
  LWADC_STRUCT_PTR lwadc_ptr
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.

Return Value

- TRUE (success)
- FALSE (failure)

Description

Waits for a new value to be available on the specified ADC input.

22.5 Data Types Used by the LWADC API

The following data types are used within the LWADC driver.

22.5.1 LWADC_INIT_STRUCT

This device-specific structure contains necessary parameters for initialization of ADC module on a particular platform.

Synopsis for MPXSxx family:

```
typedef struct lwadc_init_struct {
    uint_32 device;
    uint_32 format;
    uint_32 clock;
    uint_32 reference;
} LWADC_INIT_STRUCT, * LWADC_INIT_STRUCT_PTR;
```

Parameters

device — Device number to initialize.
format — Preset data format, see LWADC_FORMAT attribute.
clock — ADC module clock frequency.
reference — Preset reference voltage in millivolts, see LWADC_REFERENCE attribute.

22.5.2 LWADC_STRUCT

Device specific context structure keeping data for fast access to the device. A pointer to this structure is used to refer to particular ADC input in LWADC API calls.

22.5.3 Other Data Types

```
typedef enum {
    LWADC_RESOLUTION=1,
    LWADC_FREQUENCY,
    LWADC_DIVIDER,
    LWADC_DIFFERENTIAL,
    LWADC_POWER_DOWN,
    LWADC_NUMERATOR,
    LWADC_DENOMINATOR,
    LWADC_FORMAT
}  LWADC_ATTRIBUTE;
```

Members of this enum are used to refer to LWADC attributes in calls to _lwadc_set_attribute() and _lwadc_get_attribute().

The format identifiers for LWADC_FORMAT attribute are defined as macros:

LWADC_FORMAT_LEFT_JUSTIFIED

LWADC_FORMAT_RIGHT_JUSTIFIED

22.6 Example

An example application demonstrating LWADC usage is provided. The example application can be found in `\mqx\examples\lwadc` directory.

Chapter 23 Debug IO Driver

23.1 Overview

The debug I/O driver implements a data communication channel between the client and host during a debugger session. Not all debugger tools support this feature and not all support bidirectional communication. Typically the tools support output communication only (target writing to debugger console).

The driver is currently supporting the Semihost and ITM mode for Kinetis ARM Cortex M4 processors only. When a processor reaches BKPT semihost instruction, the processor enters a halt state and waits until the debugger finishes its job. Tasks and interrupts cannot be performed during the halt state. File operations (open, write, read, ioctl, close) are protected with a semaphore. The driver can be safely accessed from multiple tasks and can be set as a default "stdout" and "stdin" channel.

23.2 Source Code Location

The source code for debug driver is located in *source\io\debug* directory.

23.3 Header Files

To use a debug driver in your application, include the *bsp.h* header file, which includes the the main driver header file *debug.h*.

The file *debug_prv.h* file contains private constants and data structures which the driver uses. You should typically include this file if you change or enhance the driver itself. You may also want to look at the file as you debug your application.

23.4 Installing Drivers

The debug driver provides an installation function that either the BSP or the application calls. The function then calls **_io_dev_install_ext()** internally. Usually, the **_io_debug_install()** installation function is called from *init_bsp.c* if enabled by **BSPCFG_ENABLE_IODEBUG** configuration option in *user_config.h*

Example of the **_io_debug_install** function call:

```
#if BSPCFG_ENABLE_IODEBUG
    _io_debug_install("iodebug:", &_bsp_iodebug_init);
#endif
```

This code can be found typically in */mqx/source/bsp/init_bsp.c* file.

23.5 Initialization Record

When installing the debug driver, the pointer to initialization record is passed. The following code is an example as it can be found in *init_debug.c:*

```
const IODEBUG_INIT_STRUCT _bsp_iodebug_init = {
    IODEBUG_MODE_SEMIHOST,     /* Driver mode */
    127                        /* Length of buffered data */
    IODEBUG_NOFLUSH_CHAR       /* Default flush character */
};
```

23.6 IODEBUG_INIT_STRUCT

Synopsis

```
typedef struct {
    uint_32 MODE;
    uint_32 DATA_LENGTH;
    char    FLUSH_CHAR;
} IODEBUG_INIT_STRUCT, _PTR_ IODEBUG_INIT_STRUCT_PTR;
```

Parameters

***MODE* -** Selects the mode of operation. Available modes are:

- IODEBUG_MODE_SEMIHOST - Output is performed by "semihosting" mechanism. User messages are transferred as an exception by executing set of special instructions. During the data transfer, the processor has to be halted. Transferring each character independently can be really slow. To speed up the communication, you should use the buffer. Data is transferred together as a block.
- IODEBUG_MODE_ITM - Output is performed by using ITM, Instrumentation Trace Macrocell, one of CoreSight components. The ITM interface enables writing custom messages as trace information.

***DATA_LENGTH* -** The buffer length. The buffer will be enabled if DATA_LENGTH structure member is non zero.

***FLUSH_CHAR* -** Flush character, for instance '\n'.

23.7 Driver Services

The debug driver provides these services:

API	Calls
_io_fopen()	_io_debug_open()
_io_fclose()	_io_debug_close()
_io_read()	_io_debug_read()
_io_write()	_io_debug_write()
_io_ioctl()	_io_debug_ioctl()

23.8 Using IOCTL Commands

This section describes the I/O control commands that are used when calling **_io_ioctl()** for the IO debug driver.

23.8.1 General IOCTL commands

Command	Description	Params
IO_IOCTL_FLUSH_OUTPUT	Immediately flush ouput buffer.	NULL

23.8.2 Driver specific IOCTL commands

Command	Description	Params
IO_IOCTL_IODEBUG_SET_FLUSH_CHAR	Set flush character.	Pointer to flush character. For example: "\n"

23.9 Example

This example shows opening the debug I/O port and setting the handle as its standard output channel.

```
FILE_PTR fh_ptr;

if(NULL == (fh_ptr = fopen("iodebug:", NULL))) {
  printf("Cannot open the debug output\n");
} else {
  _io_set_handle(IO_STDOUT, fh_ptr);
  printf("This is printed to the debug output\n");
}

fflush(stdout);
if (fh_ptr != NULL) {
  fclose(fh_ptr);
}
```

Chapter 24 HMI

24.1 Overview

This section describes the HMI, Human Machine Interface, driver which is part of the MQX RTOS driver set.

The HMI driver provides an API for configuration and control of input controls such as buttons or touch electrodes and output controls such as LEDs. The HMI driver consists of two abstract layers, as shown in the figure below and described in detail in this section. The abstract and layered API can be used by various kinds of drivers implementing the human-to-machine interface.

In the current MQX version, there is one instance of the HMI driver called "BTNLED" which can be used by MQX applications to transparently handle the following input and output controls:

- Physical push buttons connected to GPIO pins and accessed with the LWGPIO driver.
- Touch electrodes handled by the Freescale Touch Sensing Library. See www.freescale.com/tss.
- LEDs connected to GPIO pins accessed with the LWGPIO driver.

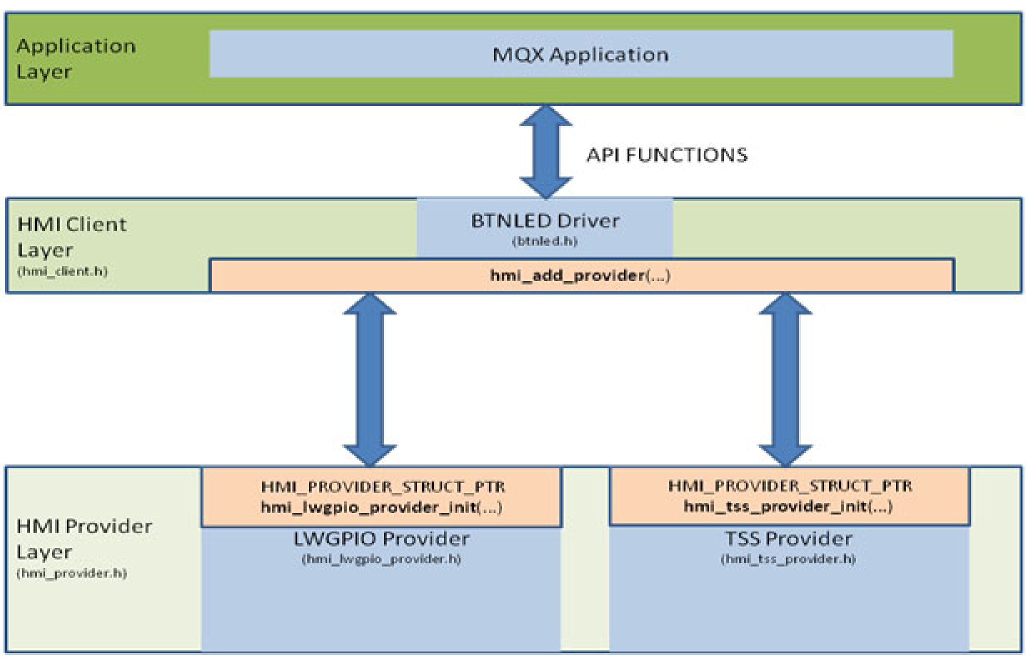


Figure 24-1. HMI Layers

HMI driver layers are designed to handle more instances of physical human-machine interfaces (so called HMI providers). For example the BTNLED driver could be extended to use PS-2 or USB Keyboards for key handling and/or external USB LED panels for signaling. The HMI API design enables such new interfaces to be implemented and used by an application without any changes in the application code.

With the HMI layered approach, there can also be other implementations of the HMI driver with a different behavior but still reusing the existing interface providers. An example of such an HMI driver instance, which is not yet implemented in the current MQX version, is a Keyboard driver which provides services of a common PC-like keyboard on top of existing providers for physical GPIO push buttons, touch electrodes, or USB keyboards.

The following sections describe the general HMI layers as well as the BTNLED driver API which can be used in the MQX applications. Use of the BTNLED driver is also demonstrated in the HMI example application located in mqx/examples/hmi folder.

The BTNLED driver and the touch sensing HMI provider driver is available for all BSPs and platforms supported by the Freescale Touch Sensing Library version available at the time of thisd release. It can be easily extended to other platforms supported by the subsequent versions of the TSS Library.

24.2 HMI Driver Layers

24.2.1 HMI Client Layer

HMI Client Layer is used as an interface between the HMI driver instance and the MQX user application. The layer provides a set of functions to manage and communicate with one or more HMI interface providers in a transparent way.

The BTNLED driver, described in more detail in the following sections, is a good example of an HMI Client. The BTNLED driver is part of the driver set in the BSP project and enables the MQX applications to access push buttons, touch electrodes, and LEDs in a transparent way independent on physical board design.

24.2.2 HMI Provider Layer

The HMI Provider Layer enables the hardware abstraction for the HMI Client Layer. This layer API enables any Client to attach to a provider and access its HMI controls in a polled or interrupt-driven way.

Two providers are implemented in the current version of MQX as follows:

- **HMI LWGPIO Provider** implements access to buttons and LEDs connected to microcontroller General-purpose I/O pins. The provider uses the MQX LWGPIO driver internally to control the GPIO ports and pins. This provider can handle both active-low and active-high push buttons and LEDs. It is typically a BSP code which initializes the provider controls according to a physical board connection.
- **TSS Provider** wraps the Freescale Touch Sensing library and enables a button-like handling of electrodes based on capacitance change detection. The TSS Library is a separate software package included in a binary form within the MQX distribution. Check the www.freescale.com/tss for the

latest version of the library, detailed documentation, and more examples of touch sensing implementation.

24.2.3 HMI UID

Each input or output control handled by the HMI Driver is identified by a 32-bit identifier called 'UID'. There is a common naming convention for general HID elements in the driver:

- **HMI_BUTTON_n** used for buttons, regardless if they are physical push buttons or touch electrodes. All MQX BSPs which implement the HMI driver assign the HMI_BUTTON_n constants sequentially:
  - HMI_BUTTON_1 assigned to the first on-board touch electrode
  - followed by other on-board touch electrodes
  - followed by physical on-board push buttons

  Some boards also enable so-called TWRPI electrode daughter cards to be attached. If this is the case, the BSP provides an API to re-map the electrodes of the selected TWRPI module to HMI_BUTTON_1..n and use them as alternatives to on-board buttons.
- **HMI_SLIDER_n** used for volume-up/down-like controls either implemented as electrical potentiometers or virtual ones based on touch position evaluation.
- **HMI_ROTARY_n** a "jog-dial" similar to the Slider control without explicit minimum and maximum position.
- **HMI_LED_n** used for LEDs or other visual on/off signals provided on board.

Internally, the UID constants are encoded as a combination of so-called usage table identifier and a usage ID, both 16-bit values.

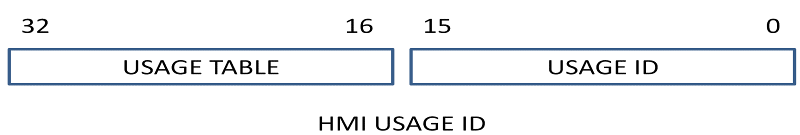


Figure 24-2. HMI Usage ID

24.3 Source Code Location

HMI driver is a part of the IO driver set and is compiled along with the other drivers into the BSP library. HMI API functions, source code, and data types are defined in source files located in the *source/io/hmi* directory.

Each BSP which makes use of the HMI driver, the BTNLED driver, implements also an *init_hmi.c* file in the BSP directory. This file contains board-specific initialization of LWGPIO and TSS providers and support for TWRPI touch-sensing electrode modules.

24.4 Header Files

To use the HMI functionality of the BTNLED driver, include the *bsp.h* header file into your application. The file contains all include statements needed for the HMI functionality.

24.5 API Function Reference

This section provides a function reference for the HMI and BTNLED drivers.

24.5.1 HMI CLIENT

24.5.1.1 hmi_client_init()

Synopsis

```
HMI_CLIENT_STRUCT_PTR hmi_client_init(void)
```

Parameters

None.

Return Value

Pointer to HMI_CLIENT_STRUCT structure used as a handle to the HMI Client instance.

Description

This function allocates and initializes memory for an instance of the HMI Client. This function is typically used internally in an initialization function of real HMI Client driver implementation - for example the BTNLED driver.

24.5.1.2 hmi_add_provider()

Synopsis

```
boolean hmi_add_provider
(
 HMI_CLIENT_STRUCT_PTR   client_handle,
 HMI_PROVIDER_STRUCT_PTR provider_handle
)
```

Parameters

client_handle [in] — Client structure handle.
provider_handle [in] — Provider structure handle.

Return Value

- TRUE - The HMI Provider has been successfully registered with the Client.
- FALSE - Failed to register the provider instance. Typically this is an out of memory issue.

Description

This function registers an HMI Provider instance in the Client's provider list. This function is typically used in the BSP initialization code to bind HMI providers to the HMI Client driver (e.g. to bind LWGPIO or TSS provider to the BTNLED driver).

24.5.1.3 hmi_remove_provider()

Synopsis

```
boolean hmi_remove_provider
(
 HMI_CLIENT_STRUCT_PTR   client_handle,
 HMI_PROVIDER_STRUCT_PTR provider_handle
)
```

Parameters

client_handle [in] — Client structure handle.
provider_handle [in] — Provider structure handle.

Return Value

- TRUE - The HMI Provider has been successfully unregistered from the Client.
- FALSE - Failed to unregister the provider. Invalid Client or Provider handle specified.

Description

This function unregisters the HMI Provider from the Client's provider list.

24.5.2 BTNLED

24.5.2.1 btnled_init()

Synopsis

```
HMI_CLIENT_STRUCT_PTR btnled_init(void)
```

Parameters

None.

Return Value

HMI_CLIENT_STRUCT_PTR handle to newly created BTNLED HMI Client.

Description

This function allocates memory for the BTNLED HMI client structure and initializes it. This function must be called first before using other driver API. The returned pointer serves as a handle to the BTNLED client instance and needs to be passed as a “handle” argument to other BTNLED driver functions.

24.5.2.2 btnled_deinit()

Synopsis

```
uint_32 btnled_deinit
```

```
(
 HMI_CLIENT_STRUCT_PTR handle
)
```

Parameters

handle [in] — Client structure handle.

Return Value

- MQX_OK

Description

This function de-allocates memory which used by the BTNLED HMI Client.

24.5.2.3 btnled_poll()

Synopsis

```
void btnled_poll
(
 HMI_CLIENT_STRUCT_PTR handle
)
```

Parameters

handle [in] — Client structure handle.

Return Value

None.

Description

This function polls all interface providers attached to the BTNLED Client. This gives each provider a chance to evaluate input signals and notify the client about any change. Each provider implements different poll functionality. For example the TSS provider invokes the TSS_Task routine and handles the TSS event callbacks.

24.5.2.4 btnled_get_value()

Synopsis

```
boolean btnled_get_value
(
 HMI_CLIENT_STRUCT_PTR handle,
 uint_32               uid,
 uint_32_ptr           value
)
```

Parameters

handle [in] — Client structure handle.
uid [in] — UID identifier of an input control.

value [out] — Returns the immediate state value for a specified.

Return Value

- TRUE - the value of the UID control has been successfully obtained.
- FALSE - the UID control is not available in any registered provider.

Description

This function gets a value of a specified UID control. For button-like controls, the returned value reflects the state of the button. It is a non-zero value when button is pressed, zero if button is released.

For a slider or rotary controls, the returned integer value reflects the current finger position within the control.

24.5.2.5 btnled_set_value()

Synopsis

```
boolean  btnled_set_value
(
 HMI_CLIENT_STRUCT_PTR handle,
 uint_32               uid,
 uint_32               value
)
```

Parameters

handle [in] — Client structure handle.
uid [in] — UID identifier of an input control.
value [in] — State value to set.

Return Value

- TRUE - the value of the UID control has been successfully set.
- FALSE - the UID control is not available in any registered provider.

Description

This function sets the control value. It is currently supported for LED output control only. Use one of HMI_VALUE_ON or HMI_VALUE_OFF constants.

24.5.2.6 btnled_toogle()

Synopsis

```
boolean btnled_toggle
(
 HMI_CLIENT_STRUCT_PTR handle,
 uint_32               uid
)
```

Parameters

handle [in] — Client structure handle.
uid [in] — UID identifier of an input control.

Return Value

- TRUE - the value of the UID control has been successfully set.
- FALSE - the UID control is not available in any registered provider.

Description

This function toggles the control value. It is currently supported for LED output control only.

24.5.2.7 btnled_add_clb()

Synopsis

```
pointer btnled_add_clb
(
 HMI_CLIENT_STRUCT_PTR handle,
 uint_32               uid,
 uint_32               state,
 void (_CODE_PTR_ function)(pointer),
 pointer               callback_parameter
)
```

Parameters

handle [in] — Client structure handle.
uid [in] — UID identifier of an input control.
state [in] — State which causes the callback to be invoked.
function [in] — Callback function to be invoked when the state changes.
callback_parameter [in] — Parameter to be passed into the callback function.

Return Value

Handle to registered callback instance. Use this value to un-register the callback function.

Description

This function registers a callback function to handle state changes for a given UID control. If the control state changes and gets equal to the selected state, the registered callback function is invoked. Use the HMI_VALUE_PUSH or HMI_VALUE_RELEASE states for button-like controls. Use HMI_VALUE_MOVEMENT state for slider and rotary controls.

24.5.2.8 btnled_remove_clb()

Synopsis

```
boolean btnled_remove_clb
(
 HMI_CLIENT_STRUCT_PTR handle,
 pointer               comp_clbreg
)
```

Parameters

handle [in] — Client structure handle.

comp_clbred [in] — Handle to registered callback instance.
state [in] — State which causes the callback to be invoked.
function [in] — Callback function to be invoked when the state changes.
callback_parameter [in] — Parameter to be passed into the callback function.

Return Value

- TRUE - The callback function has been successfully un-registered.
- FALSE - Could not un-register the callback function, invalid callback handle specified.

Description

This function un-registers the callback function previously registered to handle control state changes.

24.5.3 DATA TYPES

24.5.3.1 HMI_PROVIDER_STRUCT

A pointer to this structure represents an instance of the HMI Provider. Each provider instance is allocated and initialized by the provider-specific initialization function. The user application typically does not use this type. It is used internally by the HMI Client layer.

24.5.3.2 HMI_CLIENT_STRUCT

A pointer to this structure represents an instance of the HMI Client. Each client instance is allocated and initialized by the client-specific initialization function. The user application uses the pointer as a handle to selected HMI Client instance and passes it to all Client API functions.

24.5.3.3 HMI_TSS_INIT_STRUCT

HMI_TSS_INIT_STRUCT is the HMI TSS Provider initialization structure. An array of these structures is used in the BSP code to create and initialize any TSS provider instance. Such an initialization array should be always terminated with a zeroed structure.

The structure contains the UID identifier (UID) to be assigned to a touch sensing electrode or TSS control. The FLAG member is reserved for future use and should be zeroed.

```
typedef struct hmi_tss_init_struct
{
    uint_32     UID;
    uint_8      FLAG;
} HMI_TSS_INIT_STRUCT, _PTR_ HMI_TSS_INIT_STRUCT_PTR;
```

- For TSS Keypad controls which consist of multiple electrodes, the order of the structures in the array should match the order of electrodes configured in the TSS configuration. A HMI Provider instance for TSS Keypad control is initialized with *hmi_tss_keypad_provider_init()* function.
- For TSS Rotary and Slider controls, the initialization array typically consists of a single element assigning the UID to the whole TSS control. A TSS Provider instance for TSS Rotary control is

initialized with *hmi_tss_rotary_provider_init()* function, a provider for TSS Slider control is initialized with *hmi_tss_slider_provider_init()* function.

24.5.3.4 HMI_TSS_SYSTEM_CONTROL_STRUCT

This structure contains initial settings for the TSS Library used internally by the HMI TSS providers. The TSS Library should be initialized first by calling the *hmi_tss_init()* function before any TSS HMI provider can be created.

Refer to TSS Library User Guide for more details on configuration and electrode sensitivity parameters used in this structure.

```
typedef struct hmi_tss_system_control_struct
{
    uint_8          SYSTEM_CONFIG;
    uint_8          SYSTEM_TRIGGER;
    uint_8          NUMBER_OF_SAMPLES;
    const uint_8 _PTR_    SENSITIVITY_VALUES;
}HMI_TSS_SYSTEM_CONTROL_STRUCT, _PTR_ HMI_TSS_SYSTEM_CONTROL_STRUCT_PTR;
```

24.5.3.5 HMI_LWGPIO_INIT_STRUCT

The HMI_LWGPIO_INIT_STRUCT is the HMI LWGPIO Provider initialization structure. An array of these structures is used in the BSP code to create and initialize any instance of LWGPIO provider. The array should be always terminated with a zeroed structure.

```
typedef struct hmi_lwgpio_init_struct
{
    uint_32 UID;
    uint_32 PID;
    uint_32 FUNC;
    uint_32 FLAG;
}HMI_LWGPIO_INIT_STRUCT, _PTR_ HMI_LWGPIO_INIT_STRUCT_PTR;
```

For each button or LED, the structure binds the UID identifier with the GPIO port and pin defined by port and pin ID (PID) and LWGPIO multiplexer function setting (FUNC). The FLAG parameter can be used to define active-low or active-high pins as well as to enable internal pull-up or pull-down resistors for the button state sensing.

Refer to the *io/hmi/hmi_lwgpio_provider.h* file for more details on the supported FLAG values.

24.6 Example

The example of the HMI that demonstrates how to use HMI API functions is provided along with the MQX installation and it is located in *mqx\examples\hmi* directory.

The default settings of BTNLED client can be found in the corresponding BSP directory in the source file *init_hmi.c*.

Chapter 25 I2S Driver

25.1 Overview

This section describes the I2S device driver.

The driver uses interrupts of SSI bus for ColdFire family, or I2S peripheral module for Kinetis family and its hardware FIFO buffers.

25.2 Source Code Location

The source code for the I2S driver is located in *source\io\i2s* directory.

25.3 Header Files

To use I2S driver, include the *i2s.h, is2_audio.h* header files and the platform specific header file, *i2s_mcf54xx.h*, into your application or into the BSP header file, *bsp.h*.

25.4 Installing Drivers

The I2S driver provides installation functions that either the BSP or the application calls. **_ki2s_int_install()** function is used for Kinetis devices with the I2S peripheral module and **_mcf54xx_i2s_int_install()** function serves for mcf54xx devices. One of these functions is typically called from *init_bsp.c* if enabled by **BSPCFG_ENABLE_II2Sx** configuration option in *user_config.h*

Example of the **_mcf54xx_i2s_int_install** function call:

```
#if BSPCFG_ENABLE_II2S0
    _mcf54xx_i2s_int_install("ii2s0:", &_bsp_i2s0_init);
#endif
```

This code can be found typically in */mqx/source/bsp/<platform>/init_bsp.c* file.

After installation, the driver gets full control of SSI periphery and switches it to I2S compatible mode.

25.5 Initialization Record

When installing the I2S driver, the pointer to initialization record is passed. The following code is an example, as it can be found in *init_i2s0.c:*

```
const AUDIO_DATA_FORMAT _bsp_audio_data_init = {
   AUDIO_BIG_ENDIAN,       /* Endian of input data */
   AUDIO_ALIGNMENT_LEFT,   /* Aligment of input data */
   8,                      /* Bit size of input data */
   1,                      /* Sample size in bytes */
```

```
   1                       /* Number of channels */
};
const MCF54XX_I2S_INIT_STRUCT _bsp_i2s0_init = {
   0,                       /* I2S channel    */
   BSP_I2S0_MODE,           /* I2S mode       */
   BSP_I2S0_DATA_BITS,      /* Number of valid data bits*/
   BSP_I2S0_CLOCK_SOURCE,   /* SSI_CLOCK source*/
   FALSE,                   /* Data is stereo */
   FALSE,                   /* Transmit dummy data */
   BSP_I2S0_INT_LEVEL,      /* Interrupt level to use */
   BSP_I2S0_BUFFER_SIZE,    /* Tx buffer size */
   &_bsp_audio_data_init    /* I/O data format */
};
```

For detailed description of used data types, see chapter 25.8, “Data Types Used by the I2S Driver".

25.6 Driver Services

The I2S device driver provides the following services:

API	Calls	Description
`_io_fopen()`	`_io_i2s_open()`	Initializes hardware and calls GPIO init code to initialize peripheral and its information structure and buffers.
`_io_fclose()`	`_io_i2s_close()`	Deinitialize and close device driver. Frees all buffers and structures used by driver.
`_io_read()`	`_io_i2s_read()`	Used to transmit data over I2S bus.
`_io_write()`	`_io_i2s_write()`	Used to receive data from I2S bus.
`_io_ioctl()`	`_io_i2s_ioctl()`	Used to set/get peripheral configuration.

25.7 Using IOCTL Commands

This section describes the I/O control commands that are used when calling **_io_ioctl()** for the I2S driver.

Each of the listed functions has implemented input parameter check. If the IOCTL command requires a parameter but gets empty pointer with a NULL value, then I2S_ERROR_INVALID_PARAMETED error code is returned. When the given parameter has an incorrect range, the I2S_ERROR_PARAM_OUT_OF_RANGE error code is returned.

Command	Description	Params
IO_IOCTL_I2S_SET_MODE_MASTER	Sets driver to master mode. During switch process, processor's particular pin is configured as a master clock signal receiver or transmitter depending on whether the internal or the external source of this signal is selected.	none (NULL)
IO_IOCTL_I2S_SET_MODE_SLAVE	Sets driver to a slave mode. In this mode, device registers are set to receive clock signals such as frame sync, master clock, and bit clock from an external source.	none (NULL)
IO_IOCTL_I2S_SET_CLOCK_SOURCE_INT	Switches to internal master clock source. Clock signal is taken from the bus clock using configurable dividers. Then, according to divider settings, all derived signals setting is recomputed.	none (NULL)
IO_IOCTL_I2S_SET_CLOCK_SOURCE_EXT	Switches to external master clock source. I2S device will disconnect from the bus clcok and all bus signals must be connected externally.	none (NULL)
IO_IOCTL_I2S_SET_DATA_BITS	Sets data word length. Command parameter contains desired data length. Possible values are 8, 10, 12, 16, 18, 20, 22, or 24 bits. Data word length reduction is made by hardware so various settings have no impact on the driver's performance.	*uint_8_ptr*
IO_IOCTL_I2S_DISABLE_DEVICE	Disables I2S device. First, interrupts are disabled. After that, the whole device is disabled.	none (NULL)
IO_IOCTL_I2S_ENABLE_DEVICE	Enables I2S device. First, interrupts are enabled. After that, the whole device is enabled.	none (NULL)

Command	Description	Params
IO_IOCTL_I2S_SET_MCLK_FREQ	Sets master clock frequency. If internal clock source is selected, driver changes the master clock divider to match the requested frequency. If exact match is impossible, the closest possible master clock frequency is selected and set. This frequency is stored in the information structure from which the value can be read by the appropriate command. All other frequencies which are derived from the master clock signal are computed in a similar way. If the external clock source is selected, then its value is stored directly in the information structure as a parameter.	*uint_32_ptr*
IO_IOCTL_I2S_SET_FS_FREQ	Sets frame sync frequency. This signal is derived from the bit clock signal. The frequency is passed to the command as a parameter and internal dividers are used to set it.	*uint_32_ptr*
IO_IOCTL_I2S_TX_DUMMY_ON	Enables transmitting without input data. Transmitted data is generated by the driver and consists of 440Hz sine signal sampled with the synchronization signal frequency. The "dummy" data is stored in the driver internal memory.	none (NULL)
IO_IOCTL_I2S_TX_DUMMY_OFF	Disables transmitting without input data.	none (NULL)
IO_IOCTL_I2S_GET_MODE	Gets pointer to actual mode. Stores the mode information into a variable pointed at by the parameter. Stored value equals either I2S_MODE_SLAVE or I2S_MODE_MASTER.	*uint_8_ptr*
IO_IOCTL_I2S_GET_CLOCK_SOURCE	Gets pointer to the actual master clock source. Stores the master clock source information into a variable pointed at by the parameter. Stored value equals either I2S_CLK_INT, internal clock source, or I2S_CLK_EXT, external clock source.	*uint_8_ptr*
IO_IOCTL_I2S_GET_DATA_BITS	Copies value of actual data word length to a variable designated by the parameter.	*uint_8_ptr*
IO_IOCTL_I2S_GET_MCLK_FREQ	Copies actual master clock frequency in Hz to a variable designated by the parameter.	*uint_32_ptr*

Command	Description	Params
IO_IOCTL_I2S_GET_BCLK_FREQ	Copies actual bit clock frequency to a variable designated by the parameter. This frequency is derived from the synchronization signal frequency and the data word length. The returned frequency is in Hz.	*uint_32_ptr*
IO_IOCTL_I2S_GET_TX_DUMMY	If a device is transmitting without input data, this command copies TRUE to a variable designated by the parameter.	*boolean **
IO_IOCTL_I2S_GET_FS_FREQ	Copies the value of the frame sync signal frequency in Hz to a variable designated by the parameter.	*uint_32_ptr*
IO_IOCTL_I2S_GET_STATISTICS	Copies the actual driver I/O statistics to a variable designated by the parameter.	*I2S_STATISTICS_STRUCT_PTR*
IO_IOCTL_I2S_SET_TXFIFO_WATERMARK	Sets transmitter watermark value for both transmit FIFO buffers. When the number of samples in a buffer drops below the watermark value, an interrupt is generated. Note that setting this value can affect frequency of interrupts generated by the driver. Ensure that the watermark is not set too low. Otherwise, it could cause the buffer underflow if it is not re-filled in time. The default watermark value is set to 5. The range is 0 - 15.	*uint_8_ptr*
IO_IOCTL_I2S_SET_RXFIFO_WATERMARK	Sets the receiver watermark value for the receiver. It works the same way as a command for the transceiver, with the exception that the interrupt is generated when the number of samples in the buffer rises above the watermark value. By default, the receiver watermark value is set to 8.	*uint_8_ptr*
IO_IOCTL_I2S_GET_TXFIFO_WATERMARK	Gets the actual transmitter watermark value.	*uint_8_ptr*
IO_IOCTL_I2S_GET_RXFIFO_WATERMARK	Gets the actual receiver watermark value.	*uint_8_ptr*
IO_IOCTL_I2S_SET_CLK_ALWAYS_ENABLED_ON	Enables permanent transmission on non-data signals such as clock, synchronization and master clock signal. These signals may be used by devices connected to a bus as clock sources.	none (NULL)
IO_IOCTL_I2S_SET_CLK_ALWAYS_ENABLED_OFF	Disables permanent transmission of non-data signals.	none (NULL)

Command	Description	Params
IO_IOCTL_I2S_GET_CLK_ALWAYS_ENABLED	Gets the actual permanent transmission of non-data signal settings. If transmitting is enabled, the command returns TRUE. Otherwise, it returns FALSE.	boolean *
IO_IOCTL_I2S_CLEAR_STATISTICS	Clears statistics.	none (NULL)
IO_IOCTL_AUDIO_SET_IO_DATA_FORMAT	Sets input and output data format.	AUDIO_DATA_FORMAT_PTR
IO_IOCTL_AUDIO_GET_IO_DATA_FORMAT	Gets input and output data format.	AUDIO_DATA_FORMAT_PTR

25.8 Data Types Used by the I2S Driver

25.8.1 MCF54XX_I2S_INIT_STRUCT, KI2S_INIT_STRUCT

Synopsis

```
typedef struct mcf54xx_i2s_init_struct
{
    uint_8              CHANNEL;
    uint_8              MODE;
    uint_8              DATA_BITS;
    uint_8              CLOCK_SOURCE;
    boolean             STEREO;
    boolean             TX_DUMMY;
    _int_level          LEVEL;
    uint_32             BUFFER_SIZE;
    AUDIO_DATA_FORMAT   const * IO_FORMAT;
} MCF54XX_I2S_INIT_STRUCT, _PTR_ MCF54XX_I2S_INIT_STRUCT_PTR;


typedef struct ki2s_init_struct
{
   uint_8 CHANNEL;
   uint_8 MODE;
   uint_8 DATA_BITS;
   uint_8 CLOCK_SOURCE;
   boolean STEREO;
   boolean TX_DUMMY;
   _int_level LEVEL;
   uint_32 BUFFER_SIZE;
   AUDIO_DATA_FORMAT const * IO_FORMAT;
} KI2S_INIT_STRUCT, _PTR_ KI2S_INIT_STRUCT_PTR;
```

Parameters

***CHANNEL* -** I2S hardware channel.

***MODE* -** Driver mode (master / slave).

DATA_BITS **-** Number of valid data bits.

CLOCK_SOURCE **-** Master clock source number.

TX_DUMMY **-** Transmit without input data (yes /no).

LEVEL **-** Driver's interrupt level.

BUFFER_SIZE **-** Driver's internal buffer size.

AUDIO_DATA_FORMAT **-** Pointer to input / output data format structure.

25.8.2 I2S_STATISTICS_STRUCT

Synopsis

```
typedef struct i2s_statistics_struct
{
    uint_32 INTERRUPTS;
    uint_32 UNDERRUNS_L;
    uint_32 UNDERRUNS_R;
    uint_32 OVERRUNS_L;
    uint_32 OVERRUNS_R;
    uint_32 RX_PACKETS;
    uint_32 TX_PACKETS;
    uint_32 PACKETS_PROCESSED_L;
    uint_32 PACKETS_QUEUED_L;
    uint_32 PACKETS_REQUESTED_L;
    uint_32 PACKETS_PROCESSED_R;
    uint_32 PACKETS_QUEUED_R;
    uint_32 PACKETS_REQUESTED_R;
} I2S_STATISTICS_STRUCT, _PTR_ I2S_STATISTICS_STRUCT_PTR;
```

Parameters

INTERRUPTS **-** Number of driver generated interrupts.

UNDERRUNS_L **-** Number of left buffer underflow.

UNDERRUNS_R **-** Number of right buffer underflow.

OVERRUNS_L **-** Number of left buffer overflow.

OVERRUNS_R **-** Number of right buffer overflow.

RX_PACKETS **-** Number of samples received.

TX_PACKETS **-** Number of samples transmitted.

PACKETS_PROCESSED_L **-** Number of sent left channel samples.

PACKETS_QUEUED_L **-** Number of buffered left channel samples.

PACKETS_REQUESTED_L **-** Requested number of sent left channel samples.

PACKETS_PROCESSED_R **-** Number of sent right channel samples.

PACKETS_QUEUED_R **-** Number of buffered right channel samples.

PACKETS_REQUESTED_R - Requested number of sent right channel samples.

The content of the I2S_STATISTICS_STRUCT structure is cleared automatically when the driver is closed with _io_fclose(). It can also be cleared manually with IO_IOCTL_I2S_CLEAR_STATISTICS command.

25.8.3 AUDIO_DATA_FORMAT

Synopsis

```
typedef struct audio_data_format
{
    uint_8 ENDIAN;
    uint_8 ALIGNMENT;
    uint_8 BITS;
    uint_8 SIZE;
    uint_8 CHANNELS;
} AUDIO_DATA_FORMAT, _PTR_ AUDIO_DATA_FORMAT_PTR;
```

Parameters

ENDIAN **-** Data endianness, either AUDIO_BIG_ENDIAN, or AUDIO_LITTLE_ENDIAN.

ALIGNMENT **-** Left / right data alignment in a sample, either AUDIO_ALIGNMENT_RIGHT or AUDIO_ALIGNMENT_LEFT.

BITS **-** Data depth in bits.

SIZE **-** Data size in bytes.

CHANNELS **-** Number of channels.

25.9 Error Codes

The I2S device driver defines the following error codes.

Error code	Description
I2S_OK	Success.
I2S_ERROR_INVALID_PARAMETER	Initialization struct pointer is NULL.
I2S_ERROR_CHANNEL_INVALID	Selected channel is not available (>1).

I2S_ERROR_MODE_INVALID	MODE does not match I2S_MODE_SLAVE or I2S_MODE_MASTER.
I2S_ERROR_WORD_LENGTH_UNSUPPORTED	Invalid data word length.
I2S_ERROR_CLK_INVALID	Invalid clock source selected.
I2S_ERROR_BUFFER_SMALL	Buffer size is too small (<2).
AUDIO_ERROR_INVALID_IO_FORMAT	Invalid data format.

Chapter 26 HWTIMER Driver

26.1 Overview

This chapter describes the HWTIMER driver framework which provides a common interface for various timer modules.

The driver consists of two layers:

- Hardware specific lower layer contains implementation specifics for particular timer module. This layer is not intended for use by an application.
- Generic upper layer provides an abstraction to call the proper lower layer functions while passing a proper context structure to them. This chapter describes the generic upper layer only.

26.2 Source Code Location

The source code for HWTIMER drivers is located in `source\io\hwtimer` directory.

26.3 Header Files

To use HWTIMER driver, include the *hwtimer.h* and the device-specific *hwtimer_xxxx.h* header files from `source\io\hwtimer` in your application or in the BSP file *bsp.h*.

26.4 API Function Reference

All API functions take a pointer to caller allocated HWTIMER structure keeping the context necessary for the driver. This structure is opaque to the caller. The main purpose of the upper layer API is to provide the abstraction of the hardware specific lower layer driver.

26.4.1 hwtimer_init()

Synopsis

```
_mqx_int hwtimer_init
(
    HWTIMER_PTR hwtimer,
    const HWTIMER_DEVIF_STRUCT_PTR devif,
    uint_32 id,
    uint_32 int_priority
)
```

Parameters

hwtimer [out] — Pointer to hwtimer structure.
devif [in] — Pointer to a structure determining the lower layer.

id [in] — Numerical identifier of the timer within one timer module.
int_priority [in] — Interrupt priority.

Return Value

- MQX_OK (success)
- Error - Otherwise

Description

This function initializes caller allocated structure according to given parameters.

The device interface pointer determines low layer driver to be used. Device interface structure is exported by each low layer driver and is opaque to the applications. For details, please refer to the chapter about the low layer driver below.

The meaning of the numerical identifier varies depending on the low layer driver used. Typically, it identifies a particular timer channel to initialize.

The initialization function has to be called prior to using any other HWTIMER driver API function.

26.4.2 hwtimer_deinit()

Synopsis

```
_mqx_int hwtimer_deinit
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

Return Value

- MQX_OK (De-initialization successful)
- Error - Otherwise

Description

This function calls lower layer de-initialization function and afterwards invalidates hwtimer structure by clearing it.

26.4.3 hwtimer_set_freq()

Synopsis

```
_mqx_int hwtimer_set_freq
(
    HWTIMER_PTR hwtimer,
    uint_32     clock_id,
    uint_32     freq
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.
clock_id [in] — Clock identifier used for obtaining timer's source clock.
freq [in] — Required frequency of the timer in Hz.

Return Value

- MQX_OK (Setting frequency successful)
- Error - Otherwise

Description

This function configures the timer to tick at a frequency as closely as possible to the requested one. Actual accuracy depends on the timer module.

The function gets the value of the base frequency of the timer via the clock manager, calculates required divider ratio, and calls the low layer driver to set up the timer accordingly.

A call to this function might be consuming the CPU time as it may require complex calculation to choose the best configuration of dividers. The actual complexity depends on timer module implementation. Typically, if there is only single divider or counter preload value, there is no significant overhead.

26.4.4 hwtimer_get_freq()

Synopsis

```
uint_32 hwtimer_get_freq
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

Return Value

- Actual frequency in Hz.
- 0 - When an error occurs.

Description

The function returns the current frequency of the timer calculated from the base frequency and actual divider settings of the timer, or, if there is an error, it returns a zero.

26.4.5 hwtimer_set_period()

Synopsis

```
_mqx_int hwtimer_set_period
(
    HWTIMER_PTR hwtimer,
    uint_32     clock_id,
    uint_32     period
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.
clock_id [in] — Clock identifier used for obtaining timer's source clock.
period [in] — Required period of the timer in us.

Return Value

- MQX_OK (setting period succeeded)
- Error - Otherwise

Description

This function provides an alternate way to set up the timer to a desired period specified in microseconds rather than to a frequency in Hertz. The function gets the value of the base frequency of the timer via the clock manager, calculates required divider ratio, and calls the low layer driver to set up the timer accordingly.

A call to this function might be consuming the CPU time as it may require complex calculation to choose the best configuration of dividers. The actual complexity depends on the timer module implementation. Typically, if there is only a single divider or a counter preload value, there is no significant overhead.

26.4.6 hwtimer_get_period()

Synopsis

```
uint_32 hwtimer_get_period
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

Return Value

- Actual period in micro seconds.
- 0 - When an error occurs.

Description

This function returns the current period of the timer in microseconds, which is calculated from the base frequency, and actual divider settings of the timer.

26.4.7 hwtimer_get_modulo()

Synopsis

```
uint_32 hwtimer_get_modulo
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

Return Value

- Actual resolution (modulo) of timer.
- 0 - When an error occurs.

Description

This function returns the period of the timer in sub-ticks. It is typically called after hwtimer_set_freq() or hwtimer_set_period() to obtain actual resolution of the timer in the current configuration.

26.4.8 hwtimer_start()

Synopsis

```
_mqx_int hwtimer_start
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

Return Value

- MQX_OK (Hwtimer start successful)
- Error - Otherwise

Description

This function enables the timer and gets it running. The timer starts counting and generating interrupts each time it rolls over.

26.4.9 hwtimer_stop()

Synopsis

```
_mqx_int hwtimer_stop
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

Return Value

- MQX_OK (Hwtimer stop succeeded)
- Error - Otherwise

Description

The timer stops counting after this function is called. Pending interrupts and callbacks are canceled.

26.4.10 hwtimer_get_time()

Synopsis

```
_mqx_int hwtimer_get_time
(
    HWTIMER_PTR      hwtimer,
    HWTIMER_TIME_PTR time
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.
time [out] — Returns current value of the timer.

Return Value

- MQX_OK (Getting time succeeded)
- Error - Otherwise

Description

This function reads the current value of the timer. Elapsed periods (ticks) and current value of the timer counter (sub-ticks) are filed into the HWTIMER_TIME structure. The sub-ticks number always counts up and is reset to zero when the timer overflows regardless of the counting direction of the underlying device. The returned value corresponds to lower 32 bits of the elapsed periods (ticks).

26.4.11 hwtimer_get_ticks()

Synopsis

```
uint_32 hwtimer_get_ticks
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

Return Value

- Low 32 bits of 64 bit tick value.
- 0 - When error occurs.

Description

This function returns lower 32 bits of elapsed periods (ticks). The value is guaranteed to be obtained automatically without needing to mask the timer interrupt. The lower layer driver is not involved at all, thus a call to this function is considerably faster than hwtimer_get_time().

26.4.12 hwtimer_callback_reg()

Synopsis

```
_mqx_int hwtimer_callback_reg
(
    HWTIMER_PTR           hwtimer,
    HWTIMER_CALLBACK_FPTR callback_func,
    pointer               callback_data
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.
callback_func [in] — Function pointer to be called when the timer expires.
callback_data [in] — Arbitrary pointer passed as parameter to the callback function.

Return Value

- MQX_OK (callback registration succeeded)
- Error - Otherwise

Description

This function registers function to be called when the timer expires. The callback_data is arbitrary pointer passed as parameter to the callback function. This function must not be called from a callback routine.

26.4.13 hwtimer_callback_block()

Synopsis

```
_mqx_int hwtimer_callback_block
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

Return Value

- MQX_OK (Callback blocking succeeded)
- Error - Otherwise

Description

This function is used to block callbacks when execution of the callback function is undesired. If the timer overflows when callbacks are blocked, the callback becomes pending.

26.4.14 hwtimer_callback_unblock()

Synopsis

```
_mqx_int hwtimer_callback_unblock
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

Return Value

- MQX_OK (Callback unblocking succeeded)
- Error - Otherwise

Description

This function is used to unblock previously blocked callbacks. If there is a callback pending, it gets immediately executed. This function must not be called from a callback routine. It does not make sense to do so anyway since a callback function never gets executed while callbacks are blocked.

26.4.15 hwtimer_callback_cancel()

Synopsis

```
_mqx_int hwtimer_callback_cancel
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

Return Value

- MQX_OK (callback cancellation succeeded)
- Error - Otherwise

Description

This function cancels pending callback, if any.

26.5 Data Types Used by the HWTIMER API

The following data types are used within the HWTIMER driver.

26.5.1 HWTIMER

The context structure contains a pointer to a device interface structure, pointers to a callback function and its context, and private storage locations for arbitrary data keeping the context of the lower layer driver.

The context structure is passed to all API functions except the other parameters. The application should not access members of this structure directly.

26.5.2 HWTIMER_DEVIF_STRUCT

Each low layer driver exports an instance of this structure initialized with pointers to API functions which the driver implements. The functions should be declared as static meaning that they are not exported directly.

26.5.3 HWTIMER_TIME_STRUCT

The hwtimer time structure represents a timestamp consisting of timer elapsed periods (TICKS) and current value of the timer counter (SUBTICKS).

Synopsis

```
typedef struct hwtimer_time_struct
{
    uint_64 TICKS;
    uint_32 SUBTICKS;
} HWTIMER_TIME_STRUCT, _PTR_ HWTIMER_TIME_PTR;
```

Parameters

***TICKS* -** Ticks of timer.

***SUBTICKS* -** Subticks of timer.

26.6 Low Level Drivers Specifications

This chapter describes features related to various low level driver implementation. Currently only PIT timer module is supported. The implementation will be extended to other timer modules in the upcoming MQX releases.

26.6.1 PIT

Configarion parameters:

- **BSPCFG_HWTIMER_PIT_FREEZE** - Allows the timers to be stopped when the device enters the Debug mode. Place this configuration into the *user_config.h.* if you require this functionality of the HWTIMER driver.

26.7 Example

The example for the HWTIMER driver that shows how to use HWTIMER driver API functions is provided with the MQX installation and is located in `mqx\examples\hwtimer` directory.

There are definitions in the BSP specific header file which provide the low level device structure, BSP_HWTIMER1_DEV, with id, BSP_HWTIMER1_ID, and input frequency for the imer module, BSP_HWTIMER1_SOURCE_CLK.