Freescale MQX TM I/O Drivers

Users Guide

Document Number:
Rev. 9
04/2011

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: MQXIOUG
Rev. 9
04/2011

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 Driver

Chapter 10
I2C Driver

Chapter 11
FlashX Driver

Chapter 12
SD Card Driver

Chapter 13
RTC Driver

Chapter 14
ESDHC Driver

Chapter 15
FlexCAN Driver

Chapter 16
NAND Flash Driver

Chapter 17
DAC Driver

Chapter 18
LWGPIO 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.

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

The MQX is 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 its 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 plus 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 released by ARC at version 2.50.

MQX RTOS is a runtime library of functions that 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.

Throughout this book, we use MQX as the short name 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 in general fit into the MQX I/O model. It includes the information that apply 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)

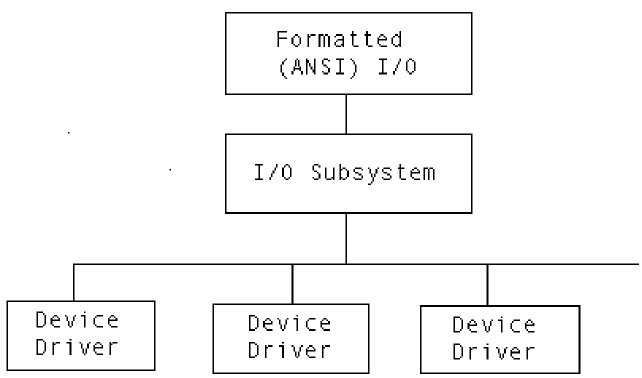

Figure 2-1. MQX I/O Layers

Due to MQX's 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 2-2 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.

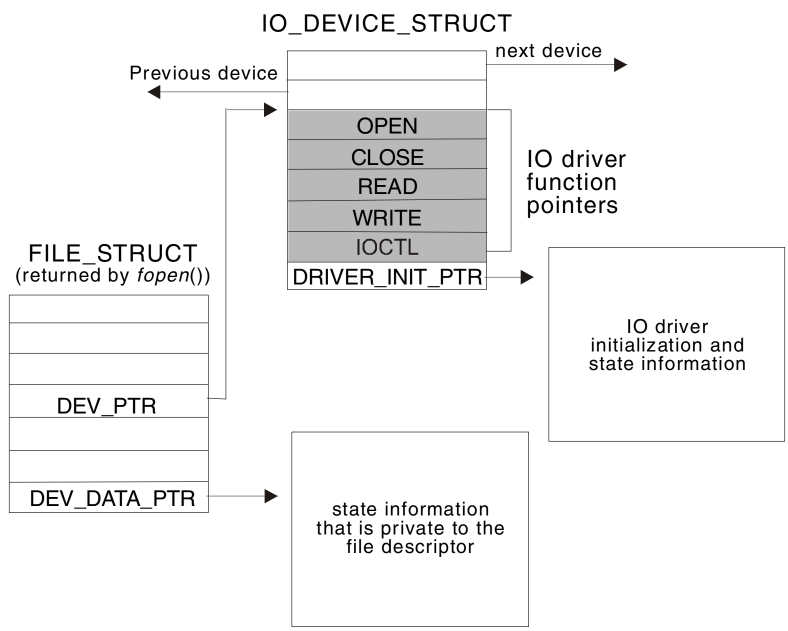

Figure 2-2. I/O Device Structure — I/O Device Drivers

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 2-3 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.

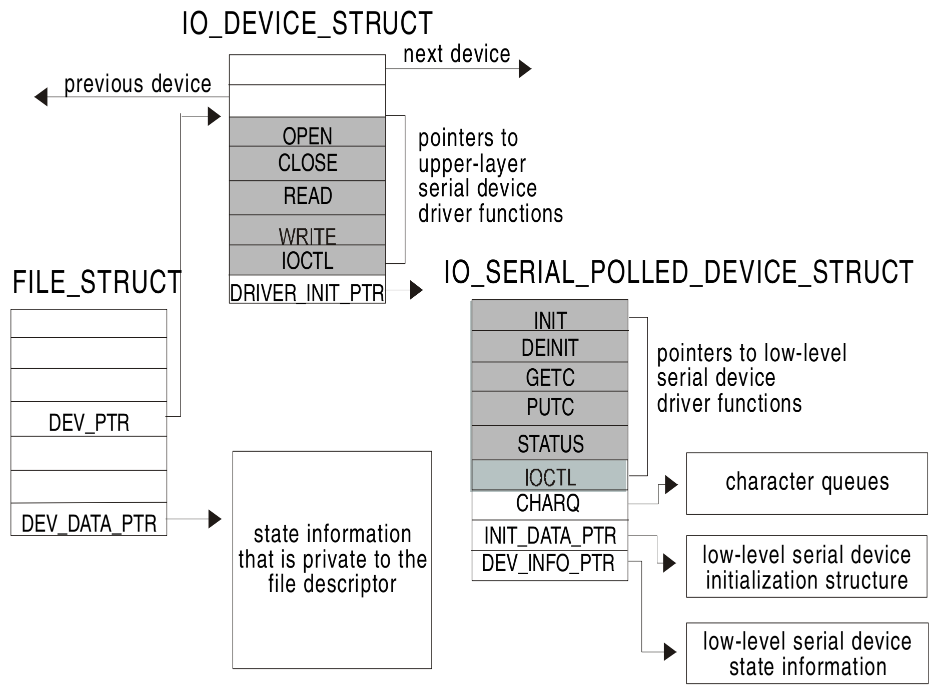

Figure 2-3. I/O Device Structure — Serial-Device Drivers

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
fgetc	_io_fgetc
fgetline	_io_fgetline

ANSI C call	MQX API
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).

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 any 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 (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 (Timer, CAN, RTC, ...) and the interrupt controller drivers implement custom API and do not follow the standard driver interface.

NOTE

At the moment of writing this manual, Freescale MQX does 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 data is available (TRUE indicates 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 apply to all serial-device drivers that accompany MQX. The subfamilies of 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 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.

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	Open the serial driver in non blocking mode. In this mode the _io_read() function doesn't wait till the receive buffer is full, but 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 transmit 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_START_BREAK	Initiate a break sequence.
IO_IOCTL_SERIAL_STOP_BREAK	Terminate a break sequence.
IO_IOCTL_SERIAL_TX_DRAINED	Return TRUE if there are no transmit characters in the FIFOs or in the software rings.
IO_IOCTL_SERIAL_DISABLE_RX	Disable or enable 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.

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.*

5.10 I/O Parity

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

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 has 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.

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 (read, write, and seek) work properly. The read and write operations are locked with a semaphore so that the entire operation can complete uninterrupted.

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 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

The GPIO driver creates hardware abstraction layer for application to 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 needed to open the GPIO device with a parameter specifying 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 *GPIO_PIN_STRUCT* ended with *GPIO_LIST_END*. To describe a pin, header file definitions must be used. 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 (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 (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

Example of *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

Pin can be used only by one file, otherwise NULL pointer is returned by **fopen**.

7.6 Driver Services

The GPIO device driver provides these 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). Array contains status of each pin using GPIO_PIN_STATUS_0 and GPIO_PIN_STATUS_1 flags.
GPIO_IOCTL_READ	Reads status of input pins and update 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.

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:

```
_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 NULL pointer is passed instead.

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	

Value	Description
ADC_CHANNEL_MEASURE_LOOP	Measurement runs continuously. The **lwevent** is set periodically after each sampling sequence (the length of sequence is specified in the *number_samples* member of ADC_INIT_CHANNEL_STRUCT). This flag is mutually exclusive with ADC_CHANNEL_MEASURE_ONCE.
ADC_CHANNEL_MEASURE_ONCE	One sequence is sampled (the length of 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.

Command	Description
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.
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 pause) measurement on specified channel file.
ADC_IOCTL_RESUME_CHANNELS	Resumes (after pause) 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 (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 (see ADLSMP bit in MCU Reference Manual). Command does not require a parameter. Command cannot be performed on channel file.

Command	Description
ADC_IOCTL_SET_HIGH_SPEED	Sets high speed conversion (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.
ADC_IOCTL_SET_LOW_SPEED	Sets high speed conversion (see ADHSC bit in MCU Reference Manual). Command does not require a parameter. Command cannot be performed on channel file.
ADC_IOCTL_SET_HW_AVERAGING	Sets averaging (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_PROCESS	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 (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 (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 (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, but 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.

Command	Description
ADC_IOCTL_SET_BASE_PERIOD	Sets period of PDB peripheral. The parameter passed to the command is the period time in microseconds.
ADC_IOCTL_TRIM_BASE_PERIOD	Similar to ADC_IOCTL_SET_BASE_PERIOD, but 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 (see the MCU Reference Manual) on PGA. Can be applied only on channels that are amplified with PGA.
ADC_IOCTL_DISABLE_CHOPPING	Disables chopping (see the MCU Reference Manual) on PGA. Can be applied only on channels that are amplified with PGA.

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 Driver

9.1 Overview

This chapter describes the SPI device driver, which is common interface for various SPI modules currently supporting ColdFire V1 SPI8 and SPI16, QSPI and DSPI. 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

9.2 Location of Source Code

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

9.3 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.

9.4 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_dspi_int_install() • _xxxx_spi8_int_install() • _xxxx_spi16_int_install()
Polled	• _xxxx_qspi_polled_install() • _xxxx_dspi_polled_install() • _xxxx_spi8_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.

9.4.1 Initialization Record

When installing the SPI device driver, the pointer to initialization record is passed. The following code is an example for the MCF51xx microcontrollers family as it can be found in the appropriate BSP code (*init_spi0.c* file). See other BSPs for similar method of installing QSPI and DSPI device drivers.

```
const MCF5XXX_SPI8_INIT_STRUCT _bsp_spi0_init = {
        0,                              /* SPI channel                   */
        MCF5XXX_SPI8_CS0,               /* Default chip select           */
        BSP_BUS_CLOCK,                  /* Bus Clock Speed               */
        BSP_SPI_BAUDRATE,               /* SPI Baud rate register value  */
        BSP_SPI_RX_BUFFER_SIZE,         /* Rx Buffer Size (interrupt only) */
        BSP_SPI_TX_BUFFER_SIZE,         /* Tx Buffer Size (interrupt only) */
        MCF51CN_INT_Vspi1,              /* Int Vector                    */
        SPI_DEVICE_MASTER_MODE,         /* Transfer mode                 */
        SPI_CLK_POL_PHA_MODE0           /* SPI clock phase               */
};
```

9.5 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()

API	Calls	
	Interrupt-driven	Polled
_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.

9.6 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.

9.7 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.

Command	Description
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.
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,

Command	Description
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.

9.8 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);
```

9.9 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.

9.10 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.

9.11 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.

9.12 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.

9.13 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 10 I^2C Driver

10.1 Overview

This chapter describes I^2C device driver. The driver includes:

- I^2C interrupt-driven I/O
- I^2C polled I/O

10.2 Source Code Location

Driver	Location
I^2C interrupt-driven	`source\io\i2c\int`
I^2C polled	`source\io\i2c\polled`

10.3 Header Files

To use an I^2C 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
I^2C interrupt-driven	• `i2c.h` • `i2c_mcfxxxx.h`
I^2C 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 I^2C device driver uses. You must include these files if you recompile an I^2C device driver. You may also want to look at the file as you debug your application.

10.4 Installing Drivers

Each I^2C 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 I^2C 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()

10.4.1 Initialization Records

When installing the I^{2}C device driver, the pointer to initialization record is passed. 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 */
};
```

10.5 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()

10.6 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 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 I^{2}C master mode.

Command	Description
IO_IOCTL_I2C_SET_SLAVE_MODE	Sets device to I^2C slave mode
IO_IOCTL_I2C_GET_MODE	Gets mode previously set.
IO_IOCTL_I2C_SET_STATION_ADDRESS	Sets device's I^2C slave address.
IO_IOCTL_I2C_GET_STATION_ADDRESS	Gets device's I^2C slave address.
IO_IOCTL_I2C_SET_DESTINATION_ADDRESS	Sets address of called device (master only).
IO_IOCTL_I2C_GET_DESTINATION_ADDRESS	Gets address of called device (master only).
IO_IOCTL_I2C_SET_RX_REQUEST	Sets in advance number of bytes 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 actual state of transmission.
IO_IOCTL_I2C_GET_STATISTICS	Gets communication statistics (structure defined in *i2c.h*.)
IO_IOCTL_I2C_CLEAR_STATISTICS	Clears 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 output buffer, waits for transfer to finish.
IO_IOCTL_I2C_GET_BUS_AVAILABILITY	Gets actual bus state (idle/busy).

10.7 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.

State	Description
I2C_STATE_LOST_ARBITRATION	Device lost arbitration, it doesn't participate on bus anymore.
I2C_STATE_FINISHED	Transmit interrupted by NACK or all requested bytes received.

10.8 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.

10.9 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.

10.10 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 11 FlashX Driver

11.1 Overview

This section contains information for the Flash device drivers that accompany the Freescale MQX.

11.2 Source Code Location

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

11.3 Header Files

To use flash drivers, include *flashx.h* and device-specific header file (for example *flash_mcf52235.h*) in your application or in the BSP file *bsp.h*.

The file *flashxprv.h* contains private constants and data structures that flash drivers use.

11.4 Hardware Supported

MQX FlashX driver enables to read and write 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/io/flashx* driver directory.

11.5 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()

11.6 Installing Drivers

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

11.7 Installing and Uninstalling Flash Devices

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

There are helper install routines available to install flashx driver for internal on-chip Flash memory named **_<CPU_name>_internal_flash_install()**. This function initializes FLASHX_INIT_STRUCT, initializes the on-chip Flash module, and calls *_io_flashx_install()*.

For external flash devices call directly **_io_flashx_install()** function with FLASHX_INIT_STRUCT parameter.

11.7.1 _io_flashx_install

Synopsis

```
_max_uint _io_flashx_install(FLASHX_INIT_STRUCT _PTR_ init_ptr)
```

Parameters

- *init_ptr* [in] — Structure containing initialization information for the flashx driver.

11.7.2 _io_flashx_uninstall

Synopsis

```
_max_uint _io_flashx_uinstall()
```

11.7.3 FLASHX_INIT_STRUCT

Synopsis

```
struct flashx_init_struct {
  char_ptr ID_PTR;
  boolean (_CODE_PTR_  SECTOR_ERASE)
     (IO_FLASHX_STRUCT_PTR, uchar_ptr, _mem_size);
  boolean (_CODE_PTR_  SECTOR_PROGRAM
     (IO_FLASHX_STRUCT_PTR, uchar_ptr, uchar_ptr,
     _mem_size);
  boolean (_CODE_PTR_   CHIP_ERASE)
     (IO_FLASHX_STRUCT_PTR);
  boolean (_CODE_PTR_   INIT)
     (IO_FLASHX_STRUCT_PTR);
  void (_CODE_PTR_        DEINIT)
     (IO_FLASHX_STRUCT_PTR);
  boolean (_CODE_PTR_   WRITE_PROTECT)
     (IO_FLASHX_STRUCT_PTR, _max_uint);
  FLASHX_BLOCK_INFO_STRUCT_PTR MAP_PTR;
  pointer                          BASE_ADDRESS;
  _mqx_uint                     WIDTH;
  _mqx_uint                      DEVICES;
  _mqx_uint                      WRITE_VERIFY;
  _mqx_int (_CODE_PTR_  IOCTL)(IO_FLASHX_STRUCT_PTR, _mqx_uint, pointer);
```

```
} FLASHX_INIT_STRUCT, _PTR_ FLASHX_INIT_STRUCT_PTR;
```

Parameters

- ID_PTR [IN] — String that identifies the device for **fopen()**.
- SECTOR_ERASE [IN] — Function to erase a flash sector.
- SECTOR_PROGRAM [IN] — Function to program a flash sector.
- CHIP_ERASE [IN] — Function to erase the entire flash.
- INIT [IN] — Function to initialize the flash device.
- DEINIT [IN] — Function to disable the flash device.
- WRITE_PROTECT [IN] — Function to disable/enable writing to the flash.
- MAP_PTR [IN] — Pointer to an array of mappings.
- BASE_ADDRESS [IN] — Base address of the device.
- WIDTH [IN] — Width of the device; one of:
  - 1 (accessed by bytes)
  - 2 (accessed by words)
  - 4 (accessed as long words)
  - 8 (accessed as double longs)
- DEVICES [IN] — Number of devices in parallel.
- WRITE_VERIFY [IN] — If true, a comparison of the original data and the flash copy is made.
- IOCTL [IN] — Optional function for device specific commands.

11.7.4 FLASHX_BLOCK_INFO_STRUCT

This structure contains information about internal flash structure. In this case, the flash contains blocks with different sector size, define each block separately. See example below. Templates with supported device definitions you can find in mqx\source\io\flashx\<producer_name>\<device_name>.c.

Synopsis

```
struct flashx_block_info_struct {
  _mqx_uint        NUM_SECTORS;
  _mem_size        START_ADDR;
  _mem_size        SECT_SIZE;
} 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 same size sectors
- SECT_SIZE [IN] — Size of the sectors in this block

Example of block info structure 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}, /* 8 x 8KB */
{ AT49BV1614A_NUM_SECTORS_2, 0x10000, AT49BV1614A_SECTOR_SIZE_2},/* 31 x 64KB */
{ 0, 0, 0 }
};
```

11.8 Internal Flash

MQX provides helper routines for internal flash driver installing and initialization. These routines fill `FLASHX_INIT_STRUCT` appropriately and call the *_io_flashx_install()* internally.

The driver is installed by calling this helper routine directly during BSP initialization, or from the application code. It is important that the `FLASHX_START_ADDR`, `FLASHX_END_ADDR` and `FLASHX_SECT_SIZE` constants are properly defined in the linker command file.

```
_mcf51cn128_internal_flash_install("flashx:", FLASHX_SIZE); /* FLASHX_SIZE must
be smaller than (FLASHX_END_ADDR - FLASHX_START_ADDR) */
```

In the example, code and linker command files distributed with MQX release are used dynamic address assignment for FlashX start address. The FlashX space starts on the first free sector in internal flash (after source code).

See linker command file code (intflash.lcf):

```
_flashx_start = __S_romp + SIZEOF(.romp);
# flashx working area spans across the whole rest of Flash memory
___FLASHX_START_ADDR = (_flashx_start + 0x03ff) / 0x400 * 0x400;
___FLASHX_END_ADDR = ___INTERNAL_FLASH_BASE + ___INTERNAL_FLASH_SIZE;
```

NOTE

If user application needs to use static memory area for FlashX driver, then new section in linker command file should be defined together with `FLASHX_START_ADDR`, `FLASHX_END_ADDR` constants pointing on start and end of that section .

11.9 External Flash

For installing driver on external flash, user should define FLASHX_INIT_STRUCTURE and call _io_flashx_install() function.

Example for installing JM28F128J3A device:

```
const FLASHX_INIT_STRUCT _bsp_flashx1_init =
{
   /* NAME           */   "flash1:",
   /* SECTOR_ERASE   */   _intel_strata_erase,
   /* SECTOR_PROGRAM */   _intel_strata_program,
   /* CHP_ERASE      */   0,
   /* INIT           */   0,
   /* DEINIT         */   0,
   /* WRITE_PROTECT  */   _intel_strata_write_protect,
   /* MAP_PTR        */   _JM28F128J3A_block_map_16bit,
```

```
   /* BASE_ADDR       */   BSP_FLASH1_BASE,
   /* WIDTH           */   BSP_FLASH1_WIDTH,
   /* DEVICES         */   BSP_FLASH1_DEVICES,
   /* WRITE VERIFY?  */   1,
   /* IOCTL               */       _io_intel_strata_ioctl
};

result = _io_flashx_install(&_bsp_flashx1_init);
```

All _intel_strata_xxx functions are device-dependent flash access routines defined in vendor-specific subdirectory in *mqx/io/flashx*.

BSP_FLASH1_BASE, BSP_FLASH1_WIDTH, BSP_FLASH1_DEVICES constant corresponds with FlexBus settings.

11.10 Device-Dependent Flash Access Routines

The driver refers to low-level functions implemented in the Flash access layer. These functions are part of the MQX release for all supported Flash memory types. User needs to provide the routines for custom memory type. The user passes pointers to these low-level functions in the `FLASHX_INIT_STRUCT` when installing the flashx driver.

The functions are located in vendor-specific subdirectory in *mqx/io/flashx.*

11.10.1 Sector Erase Function

This function is optional. It erases a sector on the device.

Synopsis

```
boolean (_CODE_PTR_  sector_erase)(
        IO_FLASHX_STRUCT_PTR handle_ptr,
        uchar_ptr            input_sect_ptr,
        mem_size             size)
```

Parameters

- *handle_ptr [IN]* — The device handle.
- *input_sect_ptr [IN]* — Address of the sector to erase.
- *size [IN]* — Amount to erase.

11.10.2 Sector Program Function

This function is required. It programs a sector on the device.

Synopsis

```
boolean (_CODE_PTR_  sector_program)(
        IO_FLASHX_STRUCT_PTR handle_ptr,
        uchar_ptr            from_ptr,
        uchar_ptr            to_ptr,
```

```
        _mem_size            size)
```

Parameters

- *handle_ptr [IN]* — The device handle
- *from_ptr [IN]* — Where to copy the data from
- *to_ptr [OUT]* — Where to copy the data to
- *size [IN]* — Size of the data to copy

11.10.3 Chip Erase Function

This function is optional. It erases the entire chip. For internal processor flash, only space from address BSPCFG_FLASHX_START to (BSPCFG_FLASHX_START + BSPCFG_FLASHX_SIZE) is erased.

Synopsis

```
boolean (_CODE_PTR_  chip_erase)(
        IO_FLASHX_STRUCT_PTR handle_ptr)
```

Parameters

- *handle_ptr [IN]* — The device handle

11.10.4 Init Function

This function is optional. It is called before opening the flashx device file.

Synopsis

```
boolean (_CODE_PTR_  init)(
        IO_FLASHX_STRUCT_PTR handle_ptr)
```

Parameters

- *handle_ptr [IN]* — The device handle.

11.10.5 DeInit Function

This function is optional. It performs any operations when no longer writing to the chip.

Synopsis

```
boolean (_CODE_PTR_  deinit)(
        IO_FLASHX_STRUCT_PTR handle_ptr)
```

Parameters

- *handle_ptr [IN]* — The device handle.

11.10.6 Write Protect Function

This function is optional. This function is called to write-enable or write-protect the device.

Synopsis

```
boolean (_CODE_PTR_  write_protect)(
```

```
IO_FLASHX_STRUCT_PTR handle_ptr,
_mqx_uint            write_protect)
```

Parameters

- *handle_ptr [IN]* — The device handle.
- *write_protect [IN]*
  - True if the device is to be write-protected.
  - False to allow writing to the device.

11.11 I/O Control Commands

This section describes the I/O control commands that you use when you call **_io_ioctl()**. Except as noted, the commands apply to all flash drivers. They are defined in *flash.h*.

Command	Description
FLASH_IOCTL_GET_BASE_ADDRESS	Base address of the flash memory.
FLASH_IOCTL_GET_NUM_SECTORS	Number of sectors in the flash device.
FLASH_IOCTL_GET_SECTOR_BASE	Start address of the current sector; that is, after performing **fseek** to a byte offset (for variable-size flash devices only).
FLASH_IOCTL_GET_SECTOR_SIZE	Sector size of the flash device.
FLASH_IOCTL_GET_WIDTH	Width of the flash device.
FLASH_IOCTL_GET_BLOCK_GROUPS	Returns the number of block groups in the device block map.
FLASH_IOCTL_GET_BLOCK_MAP	Returns the device block map address.
FLASH_IOCTL_FLUSH_BUFFER	Writes out all buffered blocks if any are dirty.
FLASH_IOCTL_ENABLE_BUFFERING	Enables RAM buffering while accessing a single Flash sector (driver internally allocates buffer of FLASH sector size). This **ioctl** can only be enabled if FLASH_IOCTL_ENABLE_SECTOR_CACHE is enabled.
FLASH_IOCTL_DISABLE_BUFFERING	Disables write-buffering.
FLASH_IOCTL_ERASE_SECTOR	Erases the specified sector.
FLASH_IOCTL_ERASE_CHIP	Erases the entire flash device.
FLASH_IOCTL_ENABLE_SECTOR_CACHE	Enables cache buffer for temporary sector backup during write access. Flash driver is able to determine if the cache is really needed (partial write access to sector containing valid data). The cache is not used when writing to a full sector, if writing to erased area only.

Command	Description
FLASH_IOCTL_DISABLE_SECTOR_CACHE	Disables the sector cache. No temporary sector buffer is allocated by the driver. Intention of this feature is RAM saving, but it restricts driver functionality - write access is only enabled in the following cases: - Incremental write to erased area - Full sector write - Partial sector overwrite when the remaining area of sector is erased. Disabling sector cache also rules out RAM buffering feature. This feature can be used e.g. if user is storing whole parameter structure still on the same place in flash, overwriting all previously stored data Then quite big RAM sector buffer is not needed.
FLASH_IOCTL_GET_WRITE_PROTECT	Returns 1 if the flash is write-protected, otherwise it returns 0.
IO_IOCTL_GET_NUM_SECTORS	Returns the number of sectors for MFS device. The default MSF_SECTOR_SIZE is 512 bytes.
IO_IOCTL_DEVICE_IDENTIFY	Returns to upper layer, what kind of device is it. It's 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.
IO_IOCTL_GET_BLOCK_SIZE	Returns the fixed MFS sector size usually 512.
FLASH_IOCTL_SWAP_FLASH_AND_RESET	Swaps the flash memory blocks. Works only with the dual flash memory controllers.
FLASH_IOCTL_WRITE_ERASE_CMD_FROM_FLASH_ENABLE	Sets up to run the low level flash write and erase routines from internal flash memory. Works only with the dual flash memory controllers.
FLASH_IOCTL_WRITE_ERASE_CMD_FROM_FLASH_DISABLE	Sets up to run the low level flash write and erase routines from RAM. Works only with the dual flash memory controllers.

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

11.12 Data Types Used with the FlexNVM

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

11.12.1 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.

11.12.2 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

11.13 Error Codes

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

Chapter 12 SD Card Driver

12.1 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 12-1.

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).

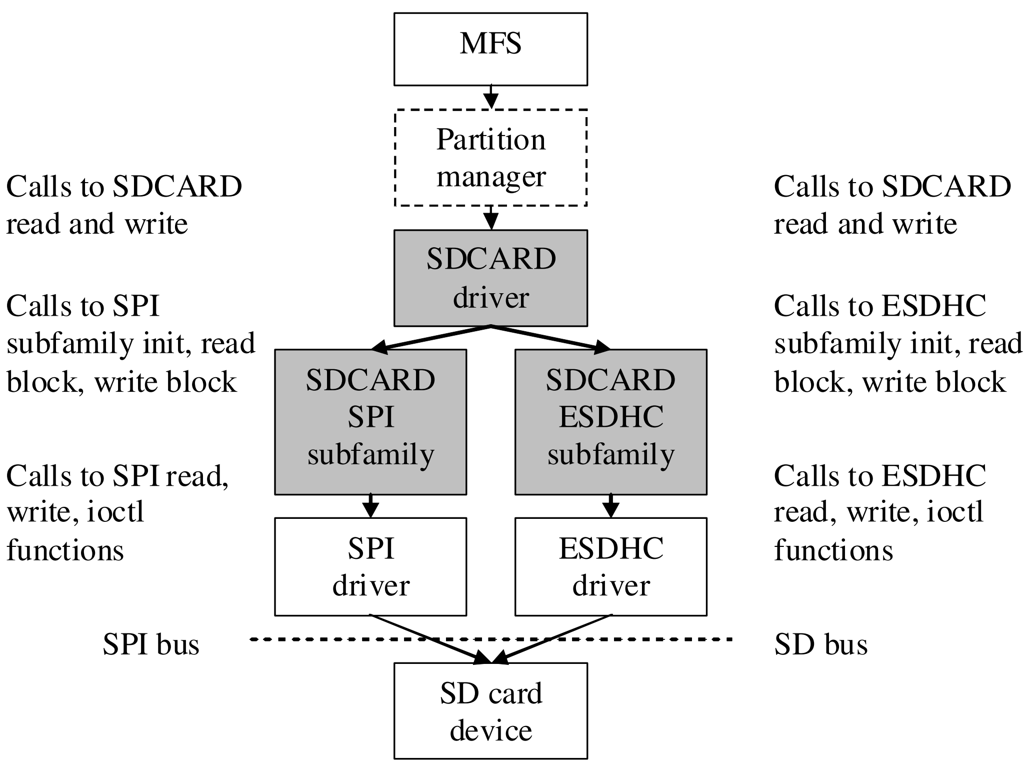

Figure 12-1. SD Card driver stack

12.2 Source Code Location

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

12.3 Header Files

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

12.4 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 low-level communication device

The default initialization structure (*_bsp_sdcard0_init*) is created in 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. In the case of 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, using GPIO pins. BSP defines `BSP_SDCARD_GPIO_DETECT` and `BSP_SDCARD_GPIO_DETECT` pins for this purpose.

12.4.1 Initialization Record

When installing the SD Card driver, the pointer to initialization record is passed. Initialization record holds pointers to low-level driver subfamily specific functions for card initialization, read block, and write block. There is also specification of chip selects or bus width used.

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
};
```

12.4.2 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, 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.

12.5 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 device with MFS.
`IO_IOCTL_GET_NUM_SECTORS`	Returns number of blocks available in SD card.This ioctl command is mandatory for using device with MFS.
`IO_IOCTL_DEVICE_IDENTIFY`	Returns flags describing SD card capabilities.This ioctl command is mandatory for using device with MFS.

12.6 Example

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

Chapter 13 RTC Driver

13.1 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).

13.2 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.

13.3 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*) into your application or into 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* into your application or into the BSP header file (*bsp.h*).

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

13.4 API Function Reference - RTC Module Related Functions

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

13.4.1 _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 run 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);
```

13.4.2 _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

13.4.3 _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");
}
```

13.4.4 _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);
```

13.4.5 _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);
```

13.4.6 _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

13.4.7 _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);
```

13.4.8 _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

13.4.9 _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);
```

13.4.10 _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

13.4.11 _rtc_set_stopwatch()

This function sets the RTC stopwatch.

Synopsis

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

Parameters

minutes [in] — Number of minutes to countdown.

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);
```

13.4.12 _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

- actual RTC minute stopwatch counter value

13.4.13 _rtc_time_to_mqx_time()

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

Return Value

- none

13.4.14 _rtc_time_from_mqx_time()

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

Return Value

- none

13.4.15 _rtc_sync_with_mqx()

This function synchronizes the 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 the RTC time
FALSE = sets the RTC time based on the MQX time

Description

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

Return Value

- MQX_OK
- RTC_INVALID_TIME, if entered date is out of MCU RTC registers range. (I.e. 1984 is the minimal year on mcf51mm.)

13.4.16 _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);
```

13.4.17 _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

13.4.18 _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);
```

13.4.19 _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

13.5 API Function Reference - IRTC Module Specific Functions

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

13.5.1 _rtc_lock()

This function locks RTC registers.

Synopsis

```
void _rtc_lock(void)
```

Parameters

none

Description

This function locks RTC registers.

Return Value

- none

13.5.2 _rtc_unlock()

This function unlocks RTC registers.

Synopsis

```
void _rtc_unlock(void)
```

Parameters

none

Description

This function unlocks RTC registers.

Return Value

- none

13.5.3 _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

13.5.4 _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

13.5.5 _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

13.5.6 _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

13.5.7 _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

13.5.8 _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

13.5.9 _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

13.5.10 _rtc_get_tamper_status()

This function is specific for 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

13.6 Data Types Used by the RTC Driver API

13.6.1 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;

}
```

13.7 Example

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

13.8 Error Codes

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

Chapter 14 ESDHC Driver

14.1 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.

14.2 Source Code Location

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

14.3 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.

14.4 Installing Driver

ESDHC device driver provides an installation function ***_xxxx_esdhc_install()*** that either the BSP or the application calls. The function then calls ***_io_dev_install_ext()*** internally. See the BSP initialization code in *init_bsp.c* for the function suitable for your hardware (xxxx in the function name). 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
_mcf5xxx_esdhc_install("esdhc:", &_bsp_esdhc0_init);
#endif
```

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

14.4.1 Initialization Record

When installing the ESDHC device driver, the pointer to initialization record is passed. The following code is an example.

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*)

14.5 Driver Services

The table below describes the ESDHC device driver services:

API	Calls	Description
`_io_fopen()`	`_mcf5xxx_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()`	`_mcf5xxx_esdhc_close()`	Resets the HW module.
`_io_read()`	`_mcf5xxx_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()`	`_mcf5xxx_esdhc_write()`	
`_io_ioctl()`	`_mcf5xxx_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.

14.6 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 (should follow successful command that sets bus width at the card). Default bus width is 1 wire.
IO_IOCTL_ESDHC_GET_BLOCK_SIZE	Returns the data transfer block size used at the host side.
IO_IOCTL_ESDHC_SET_BLOCK_SIZE	Sets the data transfer block size used at the host side (should follow successful command that sets data block size at the card). Default block size is 512 bytes.
IO_IOCTL_FLUSH_OUTPUT	Waits for HW transfer complete flag and checks errors at the host side (should be called after the whole data transfer).

14.7 Send Command Structure

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

NOTE

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

```
typedef struct esdhc_command_struct
{
        uint_8  COMMAND;
```

```
        uint_8  TYPE;
        uint_32 ARGUMENT;
        boolean READ;
        uint_32 BLOCKS;
        uint_32 RESPONSE[4];

} ESDHC_COMMAND_STRUCT, _PTR_ ESDHC_COMMAND_STRUCT_PTR;
```

Parameter	Description
COMMAND	One of the SD command definitions below.
TYPE	One of the command types below.
ARGUMENT	Command-dependant argument (argument bits must be formatted exactly according to SD specification).
READ	Sets TRUE for commands initiating data transfer from the card to the host.
BLOCKS	Number of data blocks to transfer (0 for no data transfer commands, -1 for infinite transfers).
RESPONSE	Placeholder for command response from the card (please see SD specification for details).

14.7.1 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.

Command	Description
ESDHC_CMD18	Read multiple blocks.
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.

Command	Description
ESDHC_ACMD42	Set/clear card detection.
ESDHC_ACMD51	Send SCR.

14.7.2 Command Types

This section describes the command types used in the ESDHC command structure. They are defined in *esdhc.h*.

Flag	Description
ESDHC_TYPE_NORMAL	Used with almost all commands available.
ESDHC_TYPE_SUSPEND	Instructs to release data lines (see SDIO command CMD52).
ESDHC_TYPE_RESUME	Instructs to restore data transfer (see SDIO command CMD52).
ESDHC_TYPE_ABORT	Instructs to abort data transfer (CMD12 or SDIO command CMD52).
ESDHC_TYPE_SWITCH_BUSY	Switches between response with or without busy check (see CMD6 in the ESDHC manual). This flag can be ORed with previous flags.

14.8 Card Types

This section describes the card types that are returned as 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

14.9 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)

14.10 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 (returned by IO_IOCTL_FLUSH_OUTPUT)
ESDHC_ERROR_INVALID_BUS_WIDTH	Wrong bus width detected during get/set at the host side

14.11 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.TYPE = ESDHC_TYPE_NORMAL;
command.ARGUMENT = 0;
command.READ = FALSE;
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.TYPE = ESDHC_TYPE_NORMAL;
command.ARGUMENT = 0;
command.READ = FALSE;
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.TYPE = ESDHC_TYPE_NORMAL;
command.ARGUMENT = rca;
command.READ = FALSE;
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.TYPE = ESDHC_TYPE_NORMAL;
command.ARGUMENT = rca;
command.READ = FALSE;
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.TYPE = ESDHC_TYPE_NORMAL;
command.ARGUMENT = 2;
command.READ = FALSE;
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 ();
}

/* Get current block size */
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_GET_BLOCK_SIZE, &param))
{
        _task_block ();
}
if (512 != param)
{
        _task_block ();
}

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

```
/* Read block command */
command.COMMAND = ESDHC_CMD17;
command.TYPE = ESDHC_TYPE_NORMAL;
command.ARGUMENT = sector;
command.READ = TRUE;
command.BLOCKS = 1;
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 15 FlexCAN Driver

15.1 Overview

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

15.2 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 platform supported.

15.3 Header Files

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

15.4 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.

15.4.1 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);
```

15.4.2 FLEXCAN_Start()

This function puts the FlexCAN module into 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);
```

15.4.3 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);
```

15.4.4 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);
```

15.4.5 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);
```

15.4.6 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);
```

15.4.7 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 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);
```

15.4.8 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);
```

15.4.9 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);
```

15.4.10 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);
```

15.4.11 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);
```

15.4.12 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);
```

15.4.13 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);
```

15.4.14 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);
```

15.4.15 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);
```

15.4.16 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);
```

15.4.17 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);
```

15.4.18 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);
```

15.4.19 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);
```

15.4.20 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 given message buffer. User should check 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);
```

15.4.21 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 on 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);
```

15.4.22 FLEXCAN_Request_message()

This function sets up and activates one FlexCAN message buffer to be used as 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);
```

15.4.23 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);
```

15.4.24 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 sends message once or responds to remote frame 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);
```

15.4.25 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 offset to 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);
```

15.4.26 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 offset to 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);
```

15.4.27 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);
```

15.4.28 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 response to 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);
```

15.4.29 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);
```

15.4.30 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);
```

15.4.31 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);
```

15.4.32 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);
```

15.4.33 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 same interrupt vector, so 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);
```

15.4.34 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);
```

15.4.35 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);
```

15.4.36 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);
```

15.4.37 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);
```

15.5 Data Types

This section describes the data types used by the FlexCAN driver API.

15.5.1 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;
```

15.6 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

15.7 Example

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

Chapter 16 NAND Flash Driver

16.1 Overview

This section describes the NAND Flash driver, which is used as an abstraction layer for various Nand Flash Memory devices.

16.2 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`

16.3 Header Files

To use NAND Flash driver, include *nandflash.h* and NAND Flash Controller specific header file into your application or BSP (e.g. *nfc.h*).

The *nandflashprv.h* file contains private constants and data structures that NAND Flash drivers use.

16.4 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 (sw driven solution).

MQX NAND Flash driver consists of two layers (see Figure 16-1):

- Lower Layer — It is platform dependent and has to be customized for particular NFC peripheral (or direct access). This layer implements basic NAND Flash memory operations, and have to provide API described in Section 16.6.1, "NANDFLASH_INIT_STRUCT."
- Upper Layer — It provides standard IO functionality (read, write, ioctl …). This layer can be accessed by any MQX application directly, or a File System can be mounted on the top of this layer.

User has to describe the structure of the NAND Flash memory to be supported (see Section 16.6.2, "NANDFLASH_INFO_STRUCT") and to pass this structure as an initialization parameter during driver installation, see Section 16.6.1, "NANDFLASH_INIT_STRUCT" for detail description.

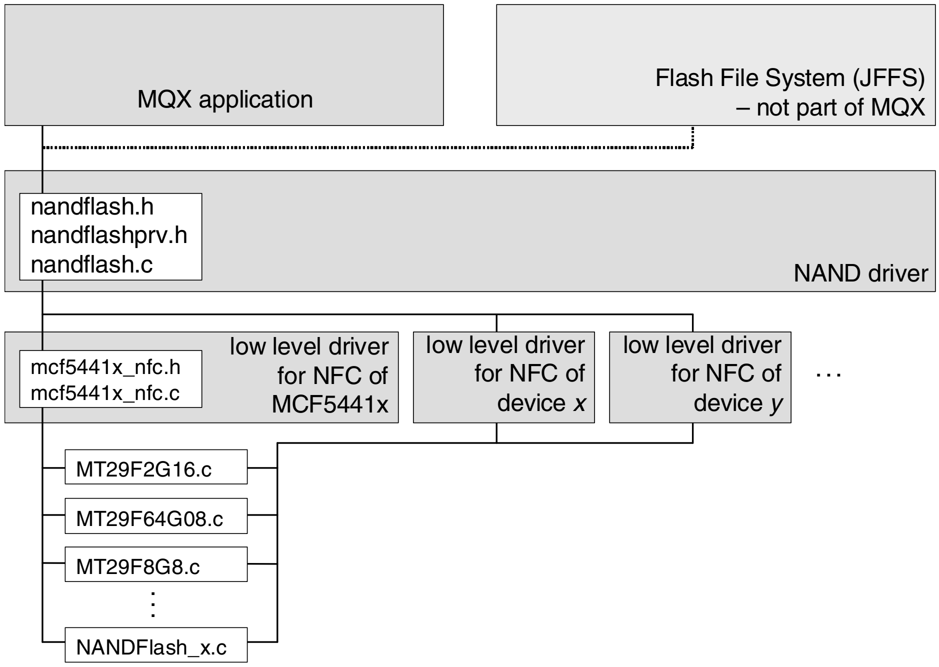

Figure 16-1. MQX NAND Flash Driver Layers

16.5 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()

16.6 Installing NAND Flash Driver

The 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.

16.6.1 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 that identifies the device for **fopen**().
- INIT— Pointer to the function that initializes the NAND flash device (low-level function).
- DEINIT— Pointer to the function that disables the NAND flash device (low-level function).
- CHIP_ERASE — Pointer to the function that erases the entire NAND flash (low-level function).
- SECTOR_ERASE— Pointer to the function that erases a flash sector (low-level function).
- BLOCK_ERASE— Pointer to the function that erases one NAND flash block (low-level function).
- PAGE_READ — Pointer to the function that reads pages of the NAND flash (low-level function).
- PAGE_PROGRAM — Pointer to the function that programs pages of the NAND flash (low-level function).

- WRITE_PROTECT — Pointer to the function that 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 that marks the defined block as bad (low-level function).
- IOCTL — Optional function for device specific commands.
- NANDFLASH_INFO_PTR — Pointer to the structure that 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 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 bsp (BSP_ECC_SIZE).
- WRITE_VERIFY — When finished programming, should a comparison of data be made to verify that the write 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.

16.6.2 NANDFLASH_INFO_STRUCT

This structure contains information about 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
};
```

16.7 NFC Peripheral Module-Specific Low Level Routines

The NAND Flash driver refers to low-level functions that implements 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*.

16.7.1 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.

16.7.2 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.

16.7.3 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.

16.7.4 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.

16.7.5 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 data to.
- *page_number [IN]* — Page number where to start reading.
- *page_count [IN]* — The amount of pages to be read.

16.7.6 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 data from.
- *page_number [IN]* — Page number where to start writing.
- *page_count [IN]* — The number of pages to be programed.

16.7.7 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.

16.7.8 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.

16.7.9 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.

16.8 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 get 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.

16.9 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.

16.10 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 erasing process failed.
NANDFLASHERR_WRITE_FAILED	Returned when writing to the NAND Flash failed.
NANDFLASHERR_TIMEOUT	Returned when any operation with the NAND Flash is time-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 17 DAC Driver

17.1 Overview

This section describes the Digital to Analog Converter (DAC) driver that accompanies the MQX release.

The DAC driver implements custom API and does not follow the standard driver interface (I/O Subsystem). Driver code is separated into Logical Device Driver (LDD) layer and Physical Device Driver (PDD) layer. This driver structure is adopted from new Processor Expert component technology, which is available for Freescale Semiconductor platforms.

17.2 Source Code Location

The source files for the DAC driver are located in `source\io\dac` directory.

17.3 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.

17.4 API Function Reference

This section serves as a function reference for the DAC module(s).

17.4.1 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 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");
}
```

17.4.2 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

17.4.3 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");
}
```

17.4.4 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);
```

17.4.5 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;
}
```

17.4.6 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.

17.4.7 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;
}
```

17.4.8 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 according 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);
```

17.4.9 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 pointer to start address and by length. First written data buffer register is defined by index, 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 one 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. So, 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 be always used.

Return Value

- DAC_ERROR_OK – OK
- DAC_ERROR_RANGE – Parameter out of range

Example

The following example shows how to write do 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);
```

17.4.10 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
);
```

17.4.11 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 to the zero directly in directly in the next trigger event.
- LDD_DAC_BUFFER_SWING_MODE – Buffer Swing Mode

  This mode is similar to the Normal mode. But when the read pointer reaches the upper limit, it does not go to the zero. It will descend by one in the next trigger events 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
);
```

17.4.12 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
);
```

17.4.13 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
);
```

17.4.14 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
);
```

17.5 Data Types Used by the DAC Driver API

17.5.1 LDD_TDeviceDataPtr

Pointer to 32-bit unsigned integer. Pointer 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;
```

17.5.2 LDD_RTOS_TDeviceDataPtr

Pointer to the structure used by RTOS containing driver-specific information. Init method receives this pointer and then passes this pointer 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.

17.5.3 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;
```

17.5.4 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;
```

17.5.5 LDD_DAC_TData

32-bit unsigned integer user data type. This data type is intended to be used for declaration of the data which will be passed to set data register methods. The size of this data type is always maximum irrespective of the current design time configuration, and may vary only across different MCU families.

Definition

```
typedef uint_32  LDD_DAC_TData;
```

17.5.6 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.

17.6 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.

17.7 Error Codes

17.7.1 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 18 LWGPIO Driver

18.1 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 I/O subsystem of MQX is initialized. LWGPIO driver is designed as a per-pin driver, meaning that LWGPIO API call handles only one pin.

18.2 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.

18.3 Header Files

To use the LWGPIO driver, include the *lwgpio.h* header file and the platform specific header file (e.g. *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*.

18.4 API Function Reference

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

18.4.1 lwgpio_init()

This function initializes structure for a GPIO pin that will be used as a pin handle in other API functions of 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 calling any other API functions 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 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 LWGPIO_DIR_OUTPUT, the corresponding level is set on GPIO's output latch (if possible, depends on a peripheral) and the pin is set to the output state. This function does not configure 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);
}
```

18.4.2 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 *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 to assign the requested functionality to the pin (GPIO mode or any other peripheral mode). The value of the *functionality* parameter represents the number stored in the multiplexer register field which selects desired functionality. For a GPIO mode, you can use pre-defined macros, which can be found in *lwgpio_ <mcu>.h* file.

Return Value

- none

Example

The following example shows how to set LWGPIO pin PTA.3 on MCF52259 MCU into the GPIO peripheral mode.

```
lwgpio_set_functionality(&led1, LWGPIO_MUX_PTA3_GPIO);
```

18.4.3 lwgpio_get_functionality()

This function gets actual pin's peripheral functionality. The pin's 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 actual pin's functionality.

Example

The following example shows how to get functionality for a pin on MCF52259 MCU.

```
func = lwgpio_get_functionality(&led1);
```

18.4.4 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 *lwgpio_init()* function.
dir [in] — One of 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 output on MCF52259.

```
lwgpio_set_direction(&led1, LWGPIO_DIR_OUTPUT);
```

18.4.5 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 *lwgpio_init()* function.
value [in] — One of 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 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);
```

18.4.6 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 *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);
```

18.4.7 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 *lwgpio_init()* function.

Description

This function is the inverse of the *lwgpio_set_value()* function. There is not always the direct relation between the physical pin state and the result of this function, because this function gets output buffer value rather than sampling pin voltage level of a pin that is set to output. To sample 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);
```

18.4.8 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 *lwgpio_init()* function.

Description

This function samples pin's signal to get 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 physical voltage level for the LWGPIO pin on MCF52259.

```
LWGPIO_VALUE value = lwgpio_get_raw(&button1);
```

18.4.9 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 an logical combination of LWGPIO_INT_xxx flags.

Description

This function prepares pin to the interrupt mode - it configures interrupt peripheral to generate interrupt flag. For most platforms, this function does not enable interrupts and it does not modify GPIO peripheral settings. If there is a need to turn pin to GPIO functionality in order to get 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 GPIO input state prior to interrupts initialization.

Return Value

- TRUE (success)
- FALSE (failure)

Example

The following example shows how to initialize 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);
}
```

18.4.10 lwgpio_int_enable()

This function enables or disables GPIO interrupts for pin on peripheral.

Synopsis

```
void lwgpio_int_enable
(
    LWGPIO_STRUCT_PTR handle,
    boolean           ena
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by *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 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 *lwgpio_int_clear_flag()* function prior to *lwgpio_int_enable()* function call.

Return Value

- none

Example

The following example shows how to enable 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);
```

18.4.11 lwgpio_int_get_flag()

This function gets the pending interrupt flag on GPIO interrupt peripheral.

Synopsis

```
boolean lwgpio_int_get_flag
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by *lwgpio_init()* function.

Description

This function returns the pin's interrupt flag on peripheral. If the interrupt is so-called keyboard interrupt, it returns 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 pending interrupt for the LWGPIO pin on MCF52259.

```
if (lwgpio_int_get_flag(&btn_int) == TRUE)
{
    /* do some action */
}
```

18.4.12 lwgpio_int_clear_flag()

This function clears the pending interrupt flag on GPIO interrupt peripheral.

Synopsis

```
void lwgpio_int_clear_flag
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

handle [in] — Pointer to the LWGPIO_STRUCT pre-initialized by *lwgpio_init()* function.

Description

This function clears the pin's interrupt flag on peripheral. If the interrupt is so-called keyboard interrupt, it clears 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);
```

18.4.13 lwgpio_int_get_vector()

This function gets the interrupt vector number that belongs to the pin or 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 *lwgpio_init()* function.

Description

This function returns 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);
```

18.5 Macro Functions Exported by the LWGPIO Driver

LWGPIO driver exports inline functions (macros) for an easy pin driving without a need 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.

18.5.1 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 (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);
```

18.5.2 lwgpio_toggle_pin_output()

This macro changes (toggles) the output value of the specified pin.

Synopsis

```
boolean inline lwgpio_toggle_pin_output(
    LWGPIO_PIN_ID id
)
```

Parameters

id [in] — LWGPIO_PIN_ID number identifying pin (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);
```

18.5.3 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 (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);
}
```

18.6 Data Types Used by the LWGPIO API

The following data types are used within the LWGPIO driver.

18.6.1 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 LWGPIO_PIN_ID value.

18.6.2 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 (heap, stack) before calling *lwgpio_init()* function.

18.6.3 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.

18.6.4 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.

18.6.5 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 different value on 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 failure status.

18.7 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.