Freescale MQX™ RTOS I/O Drivers User's Guide

Document Number: MQXIOUG
Rev. 24, 04/2015

freescale™

Contents

Section number **Title** **Page**

Chapter 1
Before You Begin

Chapter 2
MQX I/O

Chapter 3
Null-Device Driver

Chapter 4
Pipe Device Driver

Chapter 5
Simple Memory Driver

Chapter 6
Serial-Device Families

Chapter 7
GPIO Driver

Chapter 8
ADC Driver

Chapter 9
SPI Drivers

Chapter 10
Legacy SPI Drivers

Chapter 11
QSPI Drivers

Chapter 12
I2C Driver

Chapter 13
FlashX Driver

Chapter 14
SD Card Driver

Chapter 15
RTC Driver

Chapter 16
ESDHC Driver

Chapter 17
FlexCAN Driver

Chapter 18
FSL FlexCAN Driver

Chapter 19
NAND Flash Driver

Chapter 20
DAC Driver

Chapter 21
LWGPIO Driver

Chapter 22
Low Power Manager

Chapter 23
Resistive Touch Screen Driver

Chapter 24
LWADC Driver

Chapter 25
HMI

Chapter 26
Debug I/O Driver

Chapter 27
I2S Driver

Chapter 28
HWTIMER Driver

Chapter 29
FTM Quadrature Decoder Driver

Chapter 30
I/O Expander Driver

Chapter 31
DMA Driver Framework

Chapter 32
ASRC Driver Framework

Chapter 33
ESAI Driver Framework

Chapter 34
DCU4 Device Driver

Chapter 35
FBDEV Device Driver

Chapter 36
Core_mutex Driver

Chapter 1
Before You Begin

1.1 About This Book

MQX™ real-time operating system (RTOS) includes a large number of I/O device drivers, which is grouped 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 document toegether with:

- *Freescale MQX RTOS User's Guide*
- *Freescale MQX RTOS API Reference Manual*
- Driver source code

This document covers general topics, such as:

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

1.2 About MQX RTOS

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

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

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

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

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

1.3 Document Conventions

The following conventions are present throughout the document.

1.3.1 Notes

Notes point out important information. For example:

Note

Non-strict semaphores do not have priority inheritance.

1.3.2 Cautions

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

CAUTION

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

Chapter 2
MQX I/O

2.1 Overview

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

2.2 MQX I/O Layers

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

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

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

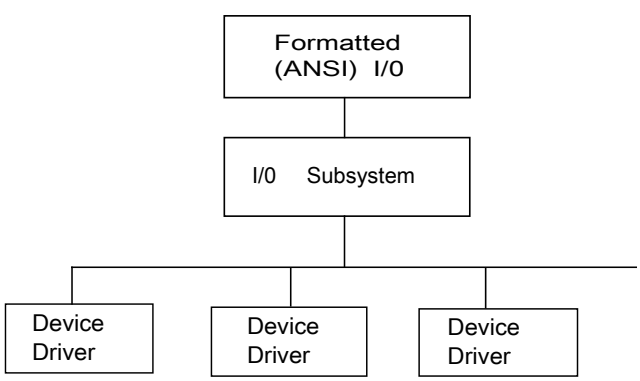

Figure 2-1. MQX I/O Layers

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

2.2.1 I/O Device Structure

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

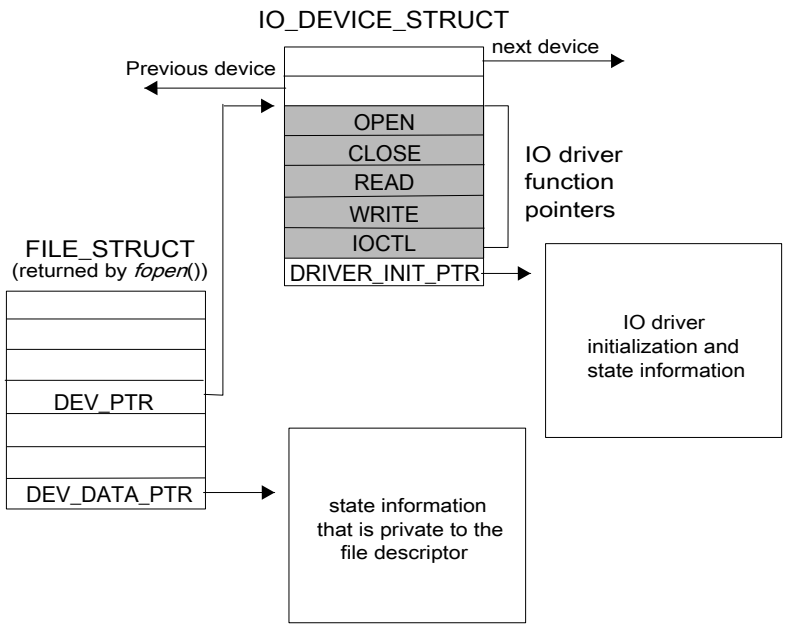

Figure 2-2. I/O Device Structure — I/O Device Drivers

2.2.2 I/O Device Structure for Serial-Device Drivers

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

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

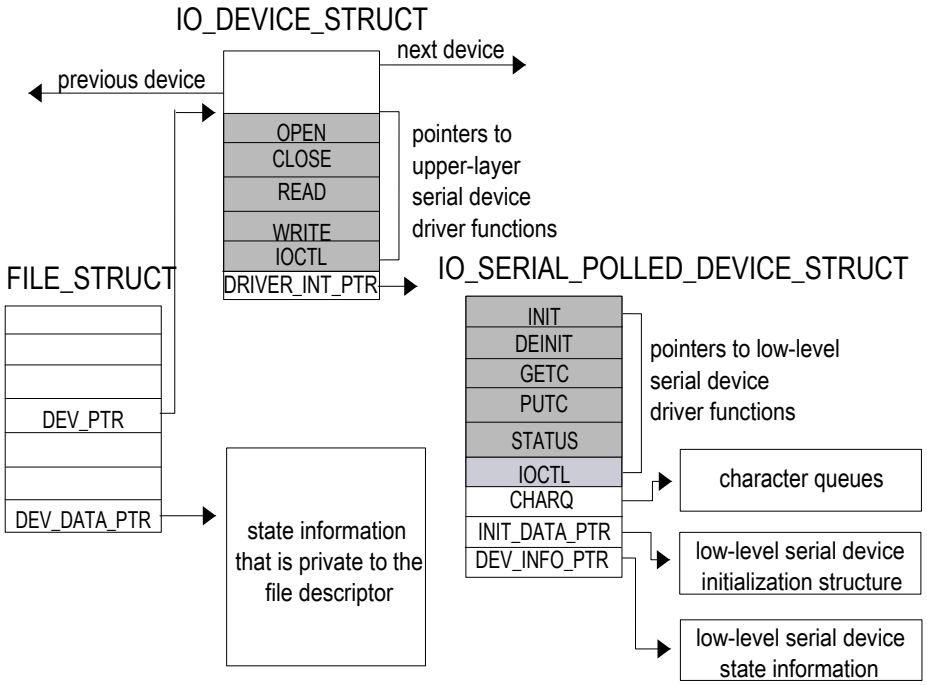

Figure 2-3. I/O Device Structure — Serial-Device Drivers

2.3 Formatted I/O Library

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

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

Table 2-1. I/O libray 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
fgets	_io_fgets
fopen	_io_fopen
fprintf	_io_fprintf
fputc	_io_fputc
fputs	_io_fputs

Table continues on the next page...

Table 2-1. I/O libray calls (continued)

ANSI C call	MQX API
fscanf	_io_fscanf
fseek	_io_fseek
fstatus	_io_fstatus
ftell	_io_ftell
fungetc	_io_fungetc
ioctl	_io_ioctl
printf	_io_printf
putc	_io_fputc
read	_io_read
scanf	_io_scanf
sprintf	_io_sprintf
sscanf	_io_sscanf
vprintf	_io_vprintf
vfprintf	_io_vfprintf
vsprintf	_io_vsprintf
write	_io_write

2.4 I/O Subsystem

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

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

- _io_dev_install
- _io_dev_install_ext
- _io_dev_uninstall
- _io_get_handle
- _io_init
- _io_set_handle

2.4.1 _io_dev_install

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

Synopsis

_mqx_uint _io_dev_install(

char *identifier,

IO_OPEN_FPTR io_open,

IO_CLOSE_FPTR io_close,

IO_READ_FPTR io_read,

IO_WRITE_FPTR io_write,

IO_IOCTL_FPTR io_ioctl,

void *io_init_data_ptr);

Parameters

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

Return Value

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

2.4.2 _io_dev_install_ext

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

Synopsis

_mqx_uint _io_dev_install(

char *identifier,

IO_OPEN_FPTR io_open,

IO_CLOSE_FPTR io_close,

IO_READ_FPTR io_read,

IO_WRITE_FPTR io_write,

IO_IOCTL_FPTR io_ioctl,

IO_UNINSTALL_FPTR io_uninstall,

void *io_init_data_ptr);

Parameters

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

Return Value

- MQX_OK (success)

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

2.4.3 _io_dev_uninstall

This function uninstalls a device dynamically.

Synopsis

_mqx_int _io_dev_uninstall(char* identifier);

Parameters

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

Return Value

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

2.4.4 _io_get_handle

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

Synopsis

void *_io_get_handle(_mqx_uint stdio_type);

Parameters

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

Return Value

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

2.4.5 _io_init

This function initializes the kernel I/O subsystem.

Synopsis

_mqx_uint _io_init(void);

Parameters

- None

Return Value

- MQX_OK (success)
- _lwsem_create function return values

2.4.6 _io_set_handle

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

Synopsis

void *_io_set_handle(

_mqx_uint stdio_type,

void *new_file_ptr);

Parameters

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

Return Value

- Previous I/O handle or NULL.

2.5 I/O Error Codes

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

2.6 I/O Device Drivers

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

2.7 Device Names

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

For example:

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

installs device `mfs1:`

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

For example:

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

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

2.8 Installing Device Drivers

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

- Call **_io_device_install()** (where **device** is replaced by the name of the driver family) from your application. Usually, the function calls **_io_dev_install()** internally to register the device with MQX RTOS. 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 RTOS.

See 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                                          *open_name_ptr,
               char                                          *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                    *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                                                    *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
  void                                              *param_ptr);
```

Parameters

- *fd_ptr [IN]* — File handle for the device.
- *cmd [IN]* — I/O control command (see 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.

Table 2-2. I/O control commands

I/O control command	Description
IO_IOCTL_CHAR_AVAIL	Check for the availability of a character.
IO_IOCTL_CLEAR_STATS	Clear the driver statistics.

Table continues on the next page...

Table 2-2. I/O control commands (continued)

I/O control command	Description
IO_IOCTL_DEVICE_IDENTIFY	Query a device to find out its properties (see 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 **uint32_t**.

The function returns the following properties in the array:

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

2.12 Error Codes

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

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

- IO_DEVICE_EXISTS

- IO_DEVICE_DOES_NOT_EXIST
- IO_ERROR_DEVICE_BUSY
- IO_ERROR_DEVICE_INVALID
- IO_ERROR_INVALID_IOCTL_CMD
- IO_ERROR_READ
- IO_ERROR_READ_ACCESS
- IO_ERROR_SEEK
- IO_ERROR_SEEK_ACCESS
- IO_ERROR_WRITE
- IO_ERROR_WRITE_ACCESS
- IO_ERROR_WRITE_PROTECTED
- IO_OK

2.13 Driver Families

MQX RTOS 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 RTOS 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 RTOS. Some of the drivers described here may not yet be supported by Freescale MQX release.

Also, not all drivers available in the Freescale MQX software are documented in this document. See the *Freescale MQX RTOS Release Notes* (document MQXRN) for the list of supported drivers.

Table 2-3. Supported driver families

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

Note

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

Note

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

Chapter 3
Null-Device Driver

3.1 Overview

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

3.2 Source Code Location

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

3.3 Header Files

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

3.4 Driver Services

The null-device driver provides the following services:

Table 3-1. Driver 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 *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 RTOS. 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:

Table 4-1. Driver 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     *identifier,
      /* [IN] The pipe queue size to use */
            uint32_t  queue_size,
      /* [IN] Currently not used */
      uint32_t  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*.

Table 4-2. I/O control commands

Command	Description	Parameters
PIPE_IOCTL_GET_SIZE	Get the size of the pipe in chars.	*param_ptr* - pointer to uint32_t
PIPE_IOCTL_FULL	Determine whether the pipe is full (TRUE indicates full).	*param_ptr* - pointer to uint32_t
PIPE_IOCTL_EMPTY	Determine whether the pipe is empty (TRUE indicates empty).	*param_ptr* - pointer to uint32_t
PIPE_IOCTL_RE_INIT	Delete all the data from the pipe.	none (NULL)
PIPE_IOCTL_CHAR_AVAIL	Determine whether the data is available (TRUE indicates that the data is available).	*param_ptr* - pointer to uint32_t
PIPE_IOCTL_NUM_CHARS_FULL	Get the number of chars in the pipe.	*param_ptr* - pointer to uint32_t
PIPE_IOCTL_NUM_CHARS_FREE	Get the amount of free chars in the pipe.	*param_ptr* - pointer to uint32_t

Chapter 5
Simple Memory Driver

5.1 Overview

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

5.2 Source Code Location

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

5.3 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.

5.4 Driver Services

The simple memory driver provides these services:

Table 5-1. Driver 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()

5.5 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      *identifier,
      /* [IN] the starting address of the device in memory */
      void      *base_address,
      /* [IN] the total size of the device */
      _file_size size
   )
```

5.6 I/O Control Commands

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

Table 5-2. I/O control commands

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.

5.7 Error Codes

No additional error codes are provided by this driver.

Chapter 6
Serial-Device Families

6.1 Overview

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

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

6.2 Source Code Location

Table 6-1. Source code location

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

6.3 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.

Table 6-2. Header files

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.

6.4 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. See the BSP initialization code in *init_bsp.c* for functions suitable for your hardware (*xxxx* in the function names below).

Table 6-3. Function names

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

6.4.1 Initialization Records

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

Synopsis for kinetis, mcf51jf and mcf51qm family

```
#include <serl_kuart.h>
typedef struct kuart_init_struct
{
   uint32_t                QUEUE_SIZE;
   uint32_t                DEVICE;
   uint32_t                CLOCK_SPEED;
   uint32_t                BAUD_RATE;
   uint32_t                RX_TX_VECTOR;
   uint32_t                ERR_VECTOR;
   uint32_t                RX_TX_PRIORITY;
```

```
   uint32_t                ERR_PRIORITY;
#if MQX_ENABLE_LOW_POWER
   CM_CLOCK_SOURCE         CLOCK_SOURCE;
   KUART_OPERATION_MODE_STRUCT_CPTR OPERATION_MODE;
#endif
} KUART_INIT_STRUCT, * KUART_INIT_STRUCT_PTR;
```

Parameters

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

DEVICE - The device to initialize.

CLOCK_SPEED - The clock speed of cpu.

BAUD_RATE - The baud rate for the channel.

RX_TX_VECTOR - RX / TX interrupt vector.

ERR_VECTOR - ERR interrupt vector.

RX_TX_PRIORITY - RX / TX interrupt vector priority.

ERR_PRIORITY - ERR interrupt vector priority.

CLOCK_SOURCE - Clock source when low power is enabled.

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

Synopsis for mcf51XX family (except mcf51jf and mcf51qm)

```
#include <serl_mcf51xx.h>
typedef struct mcf51xx_sci_init_struct
{
   uint32_t QUEUE_SIZE;
   uint32_t DEVICE;
   uint32_t CLOCK_SPEED;
   uint8_t  SCIC1_VALUE;
   uint8_t  SCIC2_VALUE;
   uint8_t  SCIC3_VALUE;
   uint32_t BAUD_RATE;
   uint32_t RX_VECTOR;
   uint32_t TX_VECTOR;
   uint32_t ER_VECTOR;
} MCF51XX_SCI_INIT_STRUCT, * MCF51XX_SCI_INIT_STRUCT_PTR;
```

Parameters

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

DEVICE - The device to initialize.

CLOCK_SPEED - The clock speed of cpu.

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

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

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

BAUD_RATE - The baud rate for the channel.

RX_VECTOR - RX interrupt vector.

TX_VECTOR - TX interrupt vector.

ER_VECTOR - ERROR interrupt vector.

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

```
#include <serl_mcf52xx.h>
typedef struct mcf52XX_uart_serial_init_struct
{
   uint32_t       QUEUE_SIZE;
   uint32_t       DEVICE;
   uint32_t       CLOCK_SPEED;
   uint32_t       VECTOR;
   _int_level    LEVEL;
   _int_priority SUBLEVEL;
   uint32_t       UMR1_VALUE;
   uint32_t       UMR2_VALUE;
   uint32_t       BAUD_RATE;
} MCF52XX_UART_SERIAL_INIT_STRUCT, * MCF52XX_UART_SERIAL_INIT_STRUCT_PTR;
```

Parameters

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

DEVICE - The device to initialize.

CLOCK_SPEED - The clock speed of cpu.

VECTOR - The interrupt vector to use if interrupt driven.

LEVEL - The interrupt level to use if interrupt driven.

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

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

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

BAUD_RATE - The baud rate for the channel.

Example

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

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

```
   /* Baud rate             */ BSPCFG_UART0_BAUD_RATE
};
```

6.5 Driver Services

The serial device driver provides these services:

Table 6-4. Driver 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()

6.6 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*.

Table 6-5. Flag values

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 \nincoming backspace outputs backspace space backspace and drops the input.
IO_SERIAL_ECHO	Echoes incoming characters.
IO_SERIAL_HW_FLOW_CONTROL	Enables hardware flow control (RTS/CTS) where available.
IO_SERIAL_NON_BLOCKING	Opens the serial driver in non blocking mode. In this mode the _io_read() function doesn't wait till the receive buffer is full. It immediately returns received characters and number of received characters.

Table continues on the next page...

Table 6-5. Flag values (continued)

Command	Description
IO_SERIAL_HW_485_FLOW_CONTROL	Enables hardware support for RS485 if it is available on target processor. Target HW automatically asserts RTS signal before transmitting the message and deasserts it after transmission is done.

6.7 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*.

Table 6-6. I/O control commands

Command	Description	Parameters
IO_IOCTL_SERIAL_CLEAR_STATS	Clear the statistics.	none (NULL)
IO_IOCTL_SERIAL_GET_BAUD	Get the BAUD rate.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_GET_CONFIG	Get the device configuration.	*param_ptr* - pointer to the serial driver initialization structure which is filled by the function (platform-specific, e.g., KUART_INIT_STRUCT for Kinetis devices)
IO_IOCTL_SERIAL_GET_FLAGS	Get the flags.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_GET_STATS	Get the statistics.	pointer to uint32_t field of size 9 that will be filled by the statistical data, see statistical information members of the KUART_INFO_STRUCT for Kinetis devices
IO_IOCTL_SERIAL_SET_BAUD	Set the BAUD rate.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_SET_FLAGS	Set the flags.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_TRANSMIT_DONE	Returns TRUE if output ring buffer empties.	*param_ptr* - pointer to bool
IO_IOCTL_SERIAL_GET_HW_SIGNAL	Returns hardware signal value.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_SET_HW_SIGNAL	Asserts the hardware signals specified.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_CLEAR_HW_SIGNAL	Clears the hardware signals specified.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_SET_DATA_BITS	Sets the number of data bits in the characters.	*param_ptr* - pointer to uint32_t

Table continues on the next page...

Table 6-6. I/O control commands (continued)

Command	Description	Parameters
IO_IOCTL_SERIAL_GET_DATA_BITS	Gets the number of data bits in the characters.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_SET_STOP_BITS	Sets the number of stop bits in the character.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_GET_STOP_BITS	Gets the number of stop bits in the character.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_DISABLE_RX	Disables or enables UART receiver.	*param_ptr* - pointer to bool
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.	none (NULL)
IO_IOCTL_SERIAL_CAN_TRANSMIT	Returns 1 in ioctl parameter when the there's a room in HW transmit buffer for another character, returns 0 otherwise.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_CAN_RECEIVE	Returns 1 in ioctl parameter when there's at least one character in input HW buffer, returns 0 otherwise.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SERIAL_GET_PARITY	Returns in ioctl parameter the type of parity that is currently configured.	*param_ptr* - pointer to uint32_t Returned value could be: IO_SERIAL_PARITY_NONE IO_SERIAL_PARITY_ODD IO_SERIAL_PARITY_EVEN
IO_IOCTL_SERIAL_SET_PARITY	Sets the given type of parity.	*param_ptr* - pointer to uint32_t The parity could be se to: IO_SERIAL_PARITY_NONE IO_SERIAL_PARITY_ODD IO_SERIAL_PARITY_EVEN

6.8 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*.

Table 6-7. I/O hardware signals

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

Table continues on the next page...

Table 6-7. I/O hardware signals (continued)

Signal	Description
IO_SERIAL_DSR	Hardware DSRsignal
IO_SERIAL_DCD	Hardware DCD signal
IO_SERIAL_RI	Hardware RI signal

6.9 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*.

Table 6-8. I/O stop-bit values

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

6.10 I/O Parity

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

Table 6-9. I/O parity values

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

6.11 RS485 Support in Serial Device

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

1. Open the serial device. If the MCU supports hardware flow control, use IO_SERIAL_HW_485_FLOW_CONTROL flag.
2. Disable transmitter if needed. This can be required if hardware echo is hardwired.
3. If the IO_SERIAL_HW_485_FLOW_CONTROL is not supported, select an appropriate GPIO pin and enable RS485 driver transmitter.
4. Send a message.
5. Wait for an empty sending queue. Use fflush().
6. Wait for the transfer complete flag. Use IO_IOCTL_SERIAL_WAIT_FOR_TC.
7. For devices without IO_SERIAL_HW_485_FLOW_CONTROL, de-assert the GPIO pin.
8. Enable receiver if it was disabled before.

Example

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

```
MQX_FILE_PTR rs485_dev = NULL;
char  data_buffer[] = "RS485 send example";
bool 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
```

6.12 Error Codes

No additional error codes are generated.

6.13 Low Power Support

The following sections describe the MQX low power functionality.

6.13.1 Overview

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

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

Both polled and interrupt serial drivers are registered at LPM during their installation. Similarly, the uninstall function unregisters them from the LPM. The registration contains callbacks that are used by the LPM to notify the serial driver about operation mode and clock configuration change. The serial peripheral device behavior is changed within these callbacks to reflect the settings defined in the `_bsp_sciX_operation_modes`

configuration structures. The configuration structures are defined for each serial peripheral channel `(sciX)` in the *init_spi.c* file. In addition to the power consumption settings, these structures also allow configuring the wakeup behavior of the serial peripheral device in the interrupt mode.

6.13.2 Data Type Definitions

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

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

```
typedef struct kuart_operation_mode_struct
{
    uint8_t FLAGS;
    uint8_t WAKEUP_BITS;
    uint8_t MA1;
    uint8_t MA2;
} KUART_OPERATION_MODE_STRUCT, * KUART_OPERATION_MODE_STRUCT_PTR;
typedef const KUART_OPERATION_MODE_STRUCT * KUART_OPERATION_MODE_STRUCT_CPTR;
```

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

Table 6-10. Low power flag values

FLAGS	Description
IO_PERIPHERAL_PIN_MUX_ENABLEIO_PERIPHERAL_PIN_MUX_DISABLE	Enables a set peripheral pins multiplexer to serial driver functionality, or disables pins associated with a particular serial peripheral device. If neither of these flags is specified, the pin multiplexer setting is not changed. If both flags are specified, the disable flag has precedence. Implementation is in the BSP specific *_bsp_serial_io_init()* function, in *init_gpio.c*.
IO_PERIPHERAL_CLOCK_ENABLE IO_PERIPHERAL_CLOCK_DISABLE	Enables/disables input clock of the serial peripheral device. If neither of these flags is specified, the pin clock setting is not changed. If both flags are specified, the disable flag has precedence. Implementation is in the BSP specific *_bsp_serial_io_init()* function, in *init_gpio.c*.
IO_PERIPHERAL_MODULE_ENABLE	If his flag is used, the serial module is enabled and internal registers are available. If it is not specified, the module is disabled.

Table continues on the next page...

Table 6-10. Low power flag values (continued)

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

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

Table 6-11. Register wakeup bits

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

The following table shows the register max bits.

Table 6-12. Register max bits

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

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

```
typedef struct kuart_init_struct
{
   uint32_t QUEUE_SIZE;
   uint32_t DEVICE;
   uint32_t CLOCK_SPEED;
   uint32_t BAUD_RATE;
   uint32_t RX_TX_VECTOR;
   uint32_t ERR_VECTOR;
   uint32_t RX_TX_PRIORITY;
   uint32_t ERR_PRIORITY;
#if MQX_ENABLE_LOW_POWER
   CM_CLOCK_SOURCE                          CLOCK_SOURCE;
   KUART_OPERATION_MODE_STRUCT_CPTR         OPERATION_MODE;
#endif

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

6.13.3 Default BSP Settings

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

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

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

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

    /* LPM_OPERATION_MODE_WAIT */
    {
```

```
        IO_PERIPHERAL_PIN_MUX_ENABLE | IO_PERIPHERAL_CLOCK_ENABLE |
IO_PERIPHERAL_MODULE_ENABLE,
        0,
        0,
        0
    },

    /* LPM_OPERATION_MODE_SLEEP */
    {
        IO_PERIPHERAL_PIN_MUX_ENABLE | IO_PERIPHERAL_CLOCK_ENABLE |
IO_PERIPHERAL_MODULE_ENABLE | IO_PERIPHERAL_WAKEUP_ENABLE |
IO_PERIPHERAL_WAKEUP_SLEEPONEXIT_DISABLE,
        0,
        0,
                0
    },
    /* LPM_OPERATION_MODE_STOP */
    {
        IO_PERIPHERAL_PIN_MUX_DISABLE | IO_PERIPHERAL_CLOCK_DISABLE,
        0,
        0,
        0
    }
};
const KUART_INIT_STRUCT _bsp_sci3_init = {
   /* queue size         */ BSPCFG_SCI3_QUEUE_SIZE,
   /* Channel            */ 3,
   /* Clock Speed        */ BSP_BUS_CLOCK,
   /* Baud rate          */ BSPCFG_SCI3_BAUD_RATE,
   /* RX/TX Int vect     */ INT_UART3_RX_TX,
   /* ERR Int vect       */ INT_UART3_ERR,
   /* RX/TX priority     */ 3,
   /* ERR priority       */ 4
#if MQX_ENABLE_LOW_POWER
   ,
   /* Clock source       */ CM_CLOCK_SOURCE_BUS,
   /* LPM operation info */ _bsp_sci3_operation_modes
#endif
};
```

6.13.4 Remarks

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

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

Chapter 7
GPIO Driver

7.1 Overview

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

The GPIO API is divided into two parts:

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

7.2 Source Code Location

Table 7-1. 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.

Table 7-2. Header files

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

7.4 Installing Drivers

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

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

7.5 Opening GPIO Device

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

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

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

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

where:

Table 7-3. GPIO pin parameter descriptions

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.

Table 7-3. GPIO pin parameter descriptions

Parameter	Description
	• GPIO_PIN_STATUS_1… for the gpio:output device; this flag sets the pin state after opening device file. • GPIO_PIN_IRQ_RISING… for the gpio:input device; this flag enables the pin status change interrupt callback function which is set by GPIO_IOCTL_SET_IRQ_FUNCTION command and allows the interrupt callback function being called when the rising edge occurs. • GPIO_PIN_IRQ_FALLING… for the gpio:input device; this flag enables the pin status change interrupt callback function which is set by GPIO_IOCTL_SET_IRQ_FUNCTION command and allows the interrupt callback function being called when the falling edge occurs. • GPIO_PIN_IRQ … this is an obsolete flag identical to the GPIO_PIN_IRQ_RISING flag.

An example of the *pin_table* initialization structure:

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

Note

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

7.6 Driver Services

The GPIO device driver provides the following services:

Table 7-4. GPIO device driver services

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

7.7 Generic I/O Control Commands

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

Table 7-5. I/O control commands

Command	Description	Parameters
GPIO_IOCTL_ADD_PINS	Adds pins to the file.	*param_ptr* - pointer to GPIO_PIN_STRUCT
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.	*param_ptr* - pointer to GPIO_PIN_STRUCT or NULL
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.	*param_ptr* - pointer to GPIO_PIN_STRUCT or NULL
GPIO_IOCTL_WRITE	Sets or clears output pins according to GPIO_PIN_STRUCT array. Pin list passed in the array must be a subset of file pins. An Array contains status of each pin by using GPIO_PIN_STATUS_0 and GPIO_PIN_STATUS_1 flags.	*param_ptr* - pointer to GPIO_PIN_STRUCT
GPIO_IOCTL_READ	Reads status of input pins and updates the GPIO_PIN_STRUCT array. Pin list passed in the array must be a subset of file pins. Uses the GPIO_PIN_STATUS mask on each item of the returned GPIO_PIN_STRUCT array to get the state of the pin.	*param_ptr* - pointer to GPIO_PIN_STRUCT
GPIO_IOCTL_SET_IRQ_FUNCTION	Sets the callback function which is invoked for any IRQ event coming from any file pin.	*param_ptr* - pointer to the IRQ function
GPIO_IOCTL_ENABLE_IRQ	Enables IRQ functionality for all IRQ pins in the file.	none (NULL)
GPIO_IOCTL_DISABLE_IRQ	Disables IRQ functionality for all IRQ pins in the file.	none (NULL)

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

Set all pins attached to the file:

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

Read pin status of input file to *read_pin_table*:

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

7.8 Hardware-Specific IOCTL Commands

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

No hardware-specific commands are implemented yet.

7.9 Error Codes

No additional error codes are generated.

Chapter 8
ADC Driver

8.1 Overview

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

8.2 Source Code Location

Table 8-1. 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:

Table 8-2. Header files

Driver	Header File
ADC driver	`adc.h`

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

8.4 Installing ADC Driver

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

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

_io_adc_install("adc1:", (void*) adc_init_struct);

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

8.4.1 Initialization Records

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

Synopsis for KADC (Kinetis family, mcf51jm and mcf51qm)

```
#include <adc_kadc.h>
typedef struct kadc_install_struct
{
  uint8_t           ADC_NUMBER;
  ADC_CLOCK_SOURCE  CLOCK_SOURCE;
  ADC_CLOCK_DIV     CLOCK_DIV;
  ADC_HSC           SPEED;
  ADC_LPC           POWER;
  uint8_t          *CALIBRATION_DATA_PTR;
  uint32_t          ADC_VECTOR;
  uint32_t          ADC_PRIORITY;
  KPDB_INIT_STRUCT const *PDB_INIT;
} KADC_INIT_STRUCT, * KADC_INIT_STRUCT_PTR;
```

Parameters

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

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

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

SPEED - High speed control. See ADC_HSC enum.

POWER - Low power control. See ADC_LPC enum.

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

ADC_VECTOR - ADC interrupt vector.

ADC_PRIORITY - Priority of the ADC interrupt.

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

Synopsis for mcf51ag, mcf51em, mcf51je and mcf51mm

Example of structure for mcf51ag:

```
#include <adc_mcf51ag.h>
typedef struct adc_install_struct
{
  uint8_t           ADC_NUMBER;
  ADC_CLOCK_SOURCE  CLOCK_SOURCE;
  ADC_CLOCK_DIV     CLOCK_DIV;
  uint8_t          *CALIBRATION_DATA_PTR;
  uint32_t          ADC_VECTOR;
} MCF51AG_ADC_INIT_STRUCT, * MCF51AG_ADC_INIT_STRUCT_PTR;
```

Parameters

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

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

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

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

ADC_VECTOR - ADC interrupt vector.

Synopsis for mcf522xx and mcf544xx

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

8.4.2 Driver Services

The ADC device driver provides these services:

Table 8-3. ADC device driver services

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

8.4.2.1 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*.

Table 8-4. Flag values

Flag Value	Description
ADC_RESOLUTION_DEFAULT	ADC native bit resolution

8.4.2.2 Opening ADC Channel File

After the ADC driver is opened and initialized as described in 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*.

Table 8-5. Constant and flag values

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

8.4.3 Using I/O Control 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.

Table 8-6. I/O control commands

Command	Description	Parameters
ADC_IOCTL_RUN_CHANNEL	Initiates measurement sequence on the specified channel file.	none (NULL)
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.	*param_ptr* - pointer to ADT_TRIGGER_MASK
ADC_IOCTL_STOP_CHANNEL	Stops measurement on specified channel file.	none (NULL)
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.	*param_ptr* - pointer to ADT_TRIGGER_MASK
ADC_IOCTL_PAUSE_CHANNEL	Pauses measurement on specified channel file.	none (NULL)
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.	*param_ptr* - pointer to ADT_TRIGGER_MASK
ADC_IOCTL_RESUME_CHANNEL	Resumes (after pausing) measurement on specified channel file.	none (NULL)
ADC_IOCTL_RESUME_CHANNELS	Resumes (after pausing) measurement on all channels assigned to a given set of triggers. The trigger mask is passed directly to **ioctl** call as an argument.	*param_ptr* - pointer to ADT_TRIGGER_MASK

8.4.3.1 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.

Table 8-7. Hardware-specific IOCTL commands

Command	Description	Parameters
ADC_IOCTL_CALIBRATE	Starts calibration process on a device. Command fails if any channel on a device is opened.	none (NULL)

Table continues on the next page...

Table 8-7. Hardware-specific IOCTL commands (continued)

Command	Description	Parameters
ADC_IOCTL_SET_CALIBRATION	Copies calibration data to the registers. Command cannot be performed on channel file.	*param_ptr* - pointer to platform-specific calibration structure (e.g., KADC_CALIB_STRUCT for Kinetis devices)
ADC_IOCTL_GET_CALIBRATION	Copies calibrated registers values to a platform-specific 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.	*param_ptr* - pointer to platform-specific calibration structure (e.g., KADC_CALIB_STRUCT for Kinetis devices)
ADC_IOCTL_SET_LONG_SAMPLE	Sets long sampling time. For more information, see ADLSMP bit in MCU Reference Manual. Number of ADC periods (2, 6, 12 or 20) is passed as a parameter to the command. Command cannot be performed on channel file.	*param_ptr* - pointer to uint32_t
ADC_IOCTL_SET_SHORT_SAMPLE	Sets short sampling time. For more information, see ADLSMP bit in MCU Reference Manual. Command does not require a parameter. Command cannot be performed on channel file.	none (NULL)
ADC_IOCTL_SET_HIGH_SPEED	Sets high speed conversion. For more information, see ADHSC bit in MCU Reference Manual. No parameter is passed to the command. Command does not require a parameter. Command cannot be performed on channel file.	none (NULL)
ADC_IOCTL_SET_LOW_SPEED	Sets high speed conversion. For more information, see ADHSC bit in MCU Reference Manual. Command does not require a parameter. Command cannot be performed on channel file.	none (NULL)
ADC_IOCTL_SET_HW_AVERAGING	Sets averaging. For more information, see AVGE bit in MCU Reference Manual. Number of samples used for averaging (0, 4, 8, 16, 32) is passed to the command as parameter. Value of zero disables averaging functionality. Command cannot be performed on channel file.	*param_ptr* - pointer to uint32_t
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.	none (NULL)
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.	none (NULL)
ADC_IOCTL_SET_OFFSET	Sets the offset for ADC. For more information, see ADCOFS register in MCU Reference Manual. The value for the register is passed as a parameter. Command cannot be performed on channel file.	*param_ptr* - pointer to uint32_t

Table continues on the next page...

Table 8-7. Hardware-specific IOCTL commands (continued)

Command	Description	Parameters
ADC_IOCTL_SET_PLUS_GAIN	Sets the plus gain for ADC. For more information, see ADCPG register in MCU Reference Manual. The value for the register is passed as a parameter. Command cannot be performed on channel file.	*param_ptr* - pointer to uint32_t
ADC_IOCTL_SET_MINUS_GAIN	Sets the minus gain for ADC. For more information, see ADCMG register in MCU Reference Manual. The value for the register is passed as a parameter. Command cannot be performed on channel file.	*param_ptr* - pointer to uint32_t
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.	*param_ptr* - pointer to uint32_t
ADC_IOCTL_SET_IDELAYREG	Similar to ADC_IOCTL_SET_IDELAY, however, the parameter passed to the command is the raw value of IDELAY register.	*param_ptr* - pointer to uint32_t
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.	*param_ptr* - pointer to PDB_INT_FCN
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.	*param_ptr* - pointer to PDB_INT_FCN
ADC_IOCTL_SET_BASE_PERIOD	Sets period of PDB peripheral. The parameter passed to the command is the period time in microseconds.	*param_ptr* - pointer to uint32_t
ADC_IOCTL_TRIM_BASE_PERIOD	Similar to ADC_IOCTL_SET_BASE_PERIOD, however, the parameter passed to the command is the raw value of MOD register.	*param_ptr* - pointer to uint32_t
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.	*param_ptr* - pointer to uint32_t
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.	*param_ptr* - pointer to uint32_t
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.	*param_ptr* - pointer to uint32_t

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:

Table 8-8. Kinetis-only IOCTL commands

Command	Description	Parameters
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.	*param_ptr* - pointer to uint32_t
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.	*param_ptr* - pointer to uint32_t
ADC_IOCTL_ENABLE_CHOPPING	Enables chopping. For more information, see the PGA Chapter in the MCU Reference Manual. Can be applied only on channels that are amplified with PGA.	none (NULL)
ADC_IOCTL_DISABLE_CHOPPING	Disables chopping. For more information, see the PGA Chapter in the Reference Manual. Can be applied only on channels that are amplified with PGA.	none (NULL)

The following table summarizes MCF51JE and MCF51MM specific IOCTL commands:

Table 8-9. MCF51JE and MCF51MM specific IOCTL commands

Command	Description	Parameters
ADC_IOCTL_PIN_DISABLE	Disable ADC functionality despite ADC channel is opened. Convenient command to allow usage of another PIN alternative with lower priority than ADC such as GPIO.	none (NULL)
ADC_IOCTL_PIN_ENABLE	Enable ADC functionality on given PIN. This command should be used for re-enabling ADC functionality temporarily disabled by ADC_IOCTL_PIN_DISABLE.	none (NULL)

8.4.4 Example

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

8.4.5 Error Codes

Table 8-10. 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:

Table 8-11. Hardware-specific error codes

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

Chapter 9
SPI Drivers

9.1 Overview

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

9.2 Location of Source Code

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

9.3 Header Files

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

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

9.4 Internal Design of SPI Drivers

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

9.5 Installing SPI Driver

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

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

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

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

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

9.5.1 Initialization Record

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

Main initialization record

```
typedef struct spi_init_struct
{
    SPI_DEVIF_STRUCT_CPTR DEVIF;
    const void           *DEVIF_INIT;
    SPI_PARAM_STRUCT      PARAMS;
    SPI_CS_CALLBACK       CS_CALLBACK;
    void                 *CS_USERDATA;
} SPI_INIT_STRUCT, * SPI_INIT_STRUCT_PTR;
```

Parameters

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

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

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

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

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

Transfer parameters record

```
typedef struct spi_param_struct
{
    uint32_t BAUDRATE;
    uint32_t MODE;
    uint32_t FRAMESIZE;
    uint32_t CS;
    uint32_t ATTR;
    uint32_t DUMMY_PATTERN;
} SPI_PARAM_STRUCT, * SPI_PARAM_STRUCT_PTR;
```

Parameters

BAUDRATE - Baud rate to use

MODE - Transfer mode (clock polarity and phase)

FRAMESIZE - Size of single SPI frame in bits

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

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

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

Example of initialization records for DSPI (Kinetis family)

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

9.6 Using the Driver

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

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

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

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

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

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

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

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

9.7 Duplex Operation

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

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

```
char buffer[11];
strcpy (buffer, "ABCDEFGHIJ");
/* ABCDEFGHIJ will be shifted to the bus and overwritten with data received */
read (spifd, buffer, 10);
fflush (spifd); /* chip select de-asserted */
```

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

```
SPI_READ_WRITE_STRUCT rw;
rw.BUFFER_LENGTH = 10;
rw.WRITE_BUFFER = (char*)send_buffer;
rw.READ_BUFFER = (char*)recv_buffer;
if (SPI_OK == ioctl (spifd, IO_IOCTL_SPI_READ_WRITE, &rw)) /*chip select asserted*/
{
  printf ("OK\n");
} else {
  printf ("ERROR\n");
}
fflush (spifd); /* chip select de-asserted */
```

9.8 Chip Selects Implemented in Software

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

- The application requires more CS signals than is supported by hardware.

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

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

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

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

9.9 I/O Open Flags

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

Table 9-1. I/O open flags

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

9.10 I/O Control Commands

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

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

Table 9-2. I/O control commands

Command	Description	Parameter
IO_IOCTL_SPI_GET_BAUD	Gets the BAUD rate.	uint32_t*
IO_IOCTL_SPI_SET_BAUD	Sets the baud rate. A supported baud rate closest to the given one is used.	uint32_t*
IO_IOCTL_SPI_GET_MODE	Gets clock polarity and phase mode.	uint32_t*
IO_IOCTL_SPI_SET_MODE	Sets clock polarity and phase mode.	uint32_t*
IO_IOCTL_SPI_GET_DUMMY_PATTERN	Gets dummy pattern for half-duplex read.	uint32_t*
IO_IOCTL_SPI_SET_DUMMY_PATTERN	Sets dummy pattern for half-duplex read.	uint32_t*
IO_IOCTL_SPI_GET_TRANSFER_MODE	Gets operation mode (master/slave).	uint32_t*
IO_IOCTL_SPI_SET_TRANSFER_MODE	Sets operation mode (master/slave).	uint32_t*
IO_IOCTL_SPI_GET_ENDIAN	Gets endian mode of the transfer.	uint32_t*
IO_IOCTL_SPI_SET_ENDIAN	Sets endian mode of the transfer.	uint32_t*
IO_IOCTL_SPI_GET_FLAGS	Gets open flags.	uint32_t*
IO_IOCTL_SPI_SET_FLAGS	Sets open flags.	uint32_t*
IO_IOCTL_SPI_GET_STATS	Gets communication statistics (structure defined in *spi.h*).	SPI_STATISTICS_STRUCT_PTR
IO_IOCTL_SPI_CLEAR_STATS	Clears communication statistics.	ignored
IO_IOCTL_FLUSH_OUTPUT	Releases the bus and de-asserts CS signals unless SPI_FLAG_NO_DEASSERT_ON_FLUSH flag is set.	ignored
IO_IOCTL_SPI_FLUSH_DEASSERT_CS	Releases the bus and de-asserts CS.	ignored
IO_IOCTL_SPI_GET_FRAMESIZE	Gets number of bits of single SPI frame.	uint32_t*
IO_IOCTL_SPI_SET_FRAMESIZE	Sets number of bits of single SPI frame.	uint32_t*
IO_IOCTL_SPI_GET_CS	Gets chip select mask.	uint32_t*
IO_IOCTL_SPI_SET_CS	Sets chip select mask.	uint32_t*
IO_IOCTL_SPI_SET_CS_CALLBACK	Sets callback function for handling CS state changes in software. Setting CS callback function possibly affects all file handles associated with the same SPI device.	SPI_CS_CALLBACK_STRUCT_PTR
IO_IOCTL_SPI_READ_WRITE	Performs simultaneous write and read full duplex operation.	SPI_READ_WRITE_STRUCT_PTR

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

9.11 Clock Modes

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

Table 9-3. Clock mode values

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

9.12 Transfer Modes

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

Table 9-4. Transfer mode values

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

9.13 Endian Mode

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

Table 9-5. Endian mode values

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

9.14 Error Codes

Following the SPI, specific error codes are defined:

Table 9-6. Error codes

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

9.15 Compatibility and Migration Guide

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

Resource sharing

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

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

By contrast, the new SPI driver framework fully supports resource sharing. There may be simultaneously open multiple file handles associated with the same SPI device. Each file handle keeps its own set of transfer parameters. In a typical use case, each file handle represents a virtual communication channel connected to a slave device. The application

then simply performs read/write/flush operations on the file handles and the SPI driver framework takes care about bus sharing and automatic reconfiguration of transfer parameters any time when this is necessary.

Backward compatibility

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

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

Error reporting

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

Chip select callback

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

Transfer statistics

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

Dummy pattern support

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

Low level initialization

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

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

Chapter 10
Legacy SPI Drivers

10.1 Overview

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

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

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

10.2 Location of Source Code

Table 10-1. Location of source code

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

10.3 Header Files

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

Table 10-2. Header files

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

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

10.4 Installing Drivers

Each SPI device driver provides an installation function that either the BSP or the application calls. The function then calls **_io_dev_install()** internally. Different installation functions exist for different SPI hardware modules. 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.

Table 10-3. Function names

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

Example of installing the QSPI device driver:

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

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

10.4.1 Initialization Record

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

Synopsis for QSPI (mcf52259)

```
#include <spi_mcf5xxx_qspi.h>
typedef struct mcf5xxx_qspi_init_struct
{
  uint32_t CHANNEL;
  uint32_t CS;
  uint32_t TRANSFER_MODE;
  uint32_t BAUD_RATE;
  uint32_t CLOCK_SPEED;
  uint32_t CLOCK_POL_PHASE;
  uint32_t RX_BUFFER_SIZE;
  uint32_t TX_BUFFER_SIZE;
} MCF5XXX_QSPI_INIT_STRUCT, * MCF5XXX_QSPI_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - SPI channel to initialize.

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

TRANSFER_MODE - SPI transfer mode (SPI_DEVICE_MASTER_MODE or SPI_DEVICE_SLAVE_MODE).

BAUD_RATE - Desired baud rate.

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

CLOCK_POL_PHASE - Clock polarity and phase (SPI_CLK_POL_PHA_MODEx).

RX_BUFFER_SIZE - Maximum size of each receive.

TX_BUFFER_SIZE - Maximum size of each transmit.

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

```
#include <spi_spi16.h>
typedef struct _spi16_init_struct
{
  uint32_t CHANNEL;
  uint32_t CS;
  uint32_t CLOCK_SPEED;
  uint32_t BAUD_RATE;
  uint32_t RX_BUFFER_SIZE;
  uint32_t TX_BUFFER_SIZE;
  uint32_t VECTOR;
  uint32_t TRANSFER_MODE;
  uint32_t CLOCK_POL_PHASE;
} SPI16_INIT_STRUCT, * SPI16_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - SPI channel to initialize.

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

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

BAUD_RATE - Desired baud rate.

RX_BUFFER_SIZE - Maximum size of each receive.

TX_BUFFER_SIZE - Maximum size of each transmit.

VECTOR - Interrupt vector to use if interrupt driven.

TRANSFER_MODE - SPI transfer mode (SPI_DEVICE_MASTER_MODE or SPI_DEVICE_SLAVE_MODE).

CLOCK_POL_PHASE - Clock polarity and phase (SPI_CLK_POL_PHA_MODEx).

10.5 Driver Services

The SPI serial device driver provides these services:

Table 10-4. Driver services

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

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

10.6 I/O Open Flags

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

Table 10-5. I/O flag values

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

10.7 I/O Control Commands

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

Table 10-6. I/O control commands

Command	Description	Parameters
IO_IOCTL_SPI_GET_BAUD	Gets the BAUD rate.	uint32_t*
IO_IOCTL_SPI_SET_BAUD	Sets the BAUD rate (finds closest to the given one).	uint32_t*
IO_IOCTL_SPI_GET_MODE	Gets clock polarity and sample mode.	uint32_t*
IO_IOCTL_SPI_SET_MODE	Sets clock polarity and sample mode.	uint32_t*
IO_IOCTL_SPI_ENABLE_MODF	Enables mode fault detection in master mode, and automatic switch to the slave mode.	none (NULL)
IO_IOCTL_SPI_DISABLE_MODF	Disables master mode fault detection.	none (NULL)
IO_IOCTL_SPI_GET_TRANSFER_MODE	Gets operation mode.	uint32_t*
IO_IOCTL_SPI_SET_TRANSFER_MODE	Sets operation mode.	uint32_t*
IO_IOCTL_SPI_GET_ENDIAN	Gets endian transfer mode.	uint32_t*
IO_IOCTL_SPI_SET_ENDIAN	Sets endian transfer mode.	uint32_t*
IO_IOCTL_SPI_DEVICE_ENABLE	Enables SPI device.	none (NULL)
IO_IOCTL_SPI_DEVICE_DISABLE	Disables SPI device.	none (NULL)
IO_IOCTL_SPI_GET_FLAGS	Gets duplex mode flags.	uint32_t*
IO_IOCTL_SPI_SET_FLAGS	Sets duplex mode flags.	uint32_t*
IO_IOCTL_SPI_GET_STATS	Gets communication statistics (structure defined in *spi.h*).	SPI_STATISTICS_STRUCT_ PTR

Table continues on the next page...

Table 10-6. I/O control commands (continued)

Command	Description	Parameters
IO_IOCTL_SPI_CLEAR_STATS	Clears communication statistics	none (NULL)
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.	none (NULL)
IO_IOCTL_SPI_FLUSH_DEASSERT_CS	Waits for transfer to finish and always deactivate CS signals regardless on opening flags.	none (NULL)
IO_IOCTL_SPI_GET_FRAMESIZE	Gets number of bits per one transfer.	uint32_t*
IO_IOCTL_SPI_SET_FRAMESIZE	Sets number of bits per one transfer.	uint32_t*
IO_IOCTL_SPI_GET_CS	Gets chip select enable mask.	uint32_t*
IO_IOCTL_SPI_SET_CS	Sets chip select enable mask.	uint32_t*
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,	SPI_CS_CALLBACK_STRUCT_PTR
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.	SPI_READ_WRITE_STRUCT_PTR
IO_IOCTL_SPI_KEEP_QSPI_CS_ACTIVE	Applies only for QSPI HW module. Modifies QSPI HW chip selects behavior. 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.	uint32_t*

10.8 Example

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

```
SPI_READ_WRITE_STRUCT rw;
rw.BUFFER_LENGTH = 10;
rw.WRITE_BUFFER = (char*)send_buffer;
rw.READ_BUFFER = (char*)recv_buffer;
printf ("READ WRITE ... ");
if (SPI_OK == ioctl (spifd, IO_IOCTL_SPI_READ_WRITE, &rw)) /*chip select asserted*/
{
  printf ("OK\n");
} else {
  printf ("ERROR\n");
}
fflush (spifd); /* chip select de-asserted */
printf ("Simultaneous write and read - EEPROM read from 0x%08x (%d):\n",
    SPI_EEPROM_ADDR1, rw.BUFFER_LENGTH);
```

10.9 Clock Modes

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

Table 10-7. Clock mode values

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

10.10 Transfer Modes

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

Table 10-8. Transfer mode values

Signal	Description
SPI_DEVICE_MASTER_MODE	Master mode (generates clock).

Table continues on the next page...

Table 10-8. Transfer mode values (continued)

Signal	Description
SPI_DEVICE_SLAVE_MODE	Slave mode.
SPI_DEVICE_BIO_MASTER_MODE	Master mode using single-wire bidirectional transfer.
SPI_DEVICE_BIO_SLAVE_MODE	Slave mode using single-wire bidirectional transfer.

10.11 Endian Transfer Modes

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

Table 10-9. Endian mode values

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

10.12 Duplex Mode Flags

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

Table 10-10. Duplex mode flag values

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

10.13 Error Codes

No additional error codes are generated.

Table 10-11. Error codes

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

Chapter 11
QSPI Drivers

11.1 Overview

This chapter describes the QSPI driver framework which provides a common interface for various QSPI modules. Currently, Vybrid QuadSPI is supported as the low level driver.

11.2 Location of Source Code

The source code for QSPI drivers is located in *mqx/source/io/qspi*.

11.3 Header Files

To use a QSPI device driver, include the header files *qspi.h* and device-specific *qspi_xxxx.h* files from *mqx/source/io/qspi* in your application or in the BSP file *bsp.h*.

The files *qspi_xxxx_prv.h* and *qspi_prv.h* contain private definitions and data structures that QSPI device drivers use. These files are required to compile a QSPI device driver. There is no need to include these files directly in your application.

11.4 Installing QSPI Driver

The installation function calls low level driver. The QSPI device driver provides an installation function that either the BSP or the application calls. The function then calls **_io_qspi_install()** internally. Usually, **_io_qspi_install()** installation function is called in *init_bsp.c*.

The following is an example of an installation of the QSPI device driver for Vybrid:

```
#if BSPCFG_ENABLE_QuadSPI0
    _io_qspi_install("qspi0:", &_bsp_quadspi0_init);
#endif
```

This code can be typically found in *mqx/source/bsp/(board_name)/init_bsp.c* file.

11.4.1 Initialization Record

The installation function requires a pointer to an initialization record to be passed to it. This record is used to initialize the device and the driver itself. The record also contains a pointer to a device interface structure which determines the low level driver to be used and pointer to its initialization data.

The description of the initialization record and related data structures follows.

11.4.2 Main Initialization Record

```
typedef struct qspi_init_struct
{
  QSPI_DEVIF_STRUCT_CPTR  DEVIF;
  const void             *DEVIF_INIT;
} QSPI_INIT_STRUCT, * QSPI_INIT_STRUCT_PTR;
```

Parameters

DEVIF - Pointer to a device interface structure defined by a particular low level driver.

DEVIF_INIT - Pointer to the initialization data specific to the low level driver.

11.4.3 Low Level Driver Record

QuadSPI driver for Vybrid defines the following low level record structure:

```
typedef struct quadspi_init_struct
{
    uint32_t                    MODULE_ID;
    QuadSPI_CLK_MODE            CLK_MODE;
    QuadSPI_IO_MODE             IO_MODE;
    uint32_t                    READ_CLK;
    uint32_t                    WRITE_CLK;
    QuadSPI_PAGE_SIZE           PAGE_SIZE;
    QuadSPI_FLASH_INFO_STRUCT *FLASH_DEVICE_INFO;
} QuadSPI_INIT_STRUCT, * QuadSPI_INIT_STRUCT_PTR;
```

MODULE_ID - module number: QuadSPI0 or QuadSPI1

CLK_MODE - clock mode: SDR or DDR mode

IO_MODE - IO mode: single pad, dual pad or quad pad.

READ_CLK - clock rate used when read from flash

WRITE_CLK - clock rate used when program flash

PAGE_SIZE - page size of specified flash

FLASH_DEVICE_INFO - Pointer to the flash device information including start address, number of sectors, and sector size. The structure definition is shown as follows:

```
typedef struct quadspi_flash_info_struct
{
    _mqx_uint START_ADDR;
    _mqx_uint NUM_SECTOR;
    _mem_size SECTOR_SIZE;
} QuadSPI_FLASH_INFO_STRUCT, * QuadSPI_FLASH_INFO_STRUCT_PTR;
```

START_ADDR - Physical block start address of the same size sectors.

NUM_SECTOR - The number of sectors with the same size in this block.

SECTOR_SIZE - The size of the sectors in this block.

11.4.4 Example of Initialization Records for QSPI

The following initialization records are used for Vybrid twrvf65gs10_a5 which is located in *mqx/source/bsp/twrvf65gs10_a5/init_qspi.c*.

```
static QuadSPI_FLASH_INFO_STRUCT _bsp_quadspi0_flash_device[] = {
    {
        0,
        64,
        0x40000
    }, {
        0x1000000,
        0,
        0
    }, {
        0x1000000,
        64,
        0x40000
    }, {
        0x2000000,
        0,
        0
   }
};
static const QuadSPI_INIT_STRUCT _bsp_quadspi0_init_data = {
    0,                          /* QSPI controller id */
    QuadSPI_CLK_DDR_MODE,       /* Clock Mode */
    QuadSPI_SINGLE_MODE,         /* IO Mode */
    33000000,                    /* Serial Clock: Read */
    33000000,                    /* Serial Clock: Write */
    QuadSPI_PAGE_256,            /* Page size */
    _bsp_quadspi0_flash_device /* flash device information */
};
const QSPI_INIT_STRUCT _bsp_quadspi0_init = {
    &_qspi_quadspi_devif,       /* Low level driver interface */
    &_bsp_quadspi0_init_data,  /* Low level driver init data */
};
```

11.5 Using the Driver

The QSPI device driver provides the following functions:

Table 11-1. QSPI Driver Functions

ANSI C Call	MQX API	Driver Calls
fopen	_io_fopen()	_io_qspi_open
fclose	_io_fclose()	_io_qspi_close
fread	_io_read()	_io_qspi_read
fwrite	_io_write()	_io_qspi_write
ioctl	_io_ioctl()	_io_qspi_ioctl

11.5.1 Open/Close QSPI Device

A file handle to the QSPI device is obtained by the **_io_open()** API call and released by the **_io_fclose()** API call. Note that the file name of qspi open string should match the file name which was installed in the BSP.

```
qspifd = fopen("qspi0:", NULL); /* open qspi0 */
fclose (qspifd);                /* close qspi0 */
```

11.5.2 Using IOCTL Commands

This section describes the I/O control commands used in **_io_ioctl()** for a particular QSPI device driver. They are defined in *qspi.h* and *qspi_xxxx.h* for the low level driver.

The supported IOCTL commands are shown in the following table.

Table 11-2. Supported IOCTRL Commands

Command	Description	Parameters
QuadSPI_IOCTL_SET_SDR	Set QuadSPI controller to SDR mode	none (NULL)
QuadSPI_IOCTL_SET_DDR	Set QuadSPI controller to DDR mode	none (NULL)
QuadSPI_IOCTL_SET_READ_SPEED	Set QuadSPI read speed (clock rate)	*param_ptr*- pointer to uint32_t
QuadSPI_IOCTL_SET_WRITE_SPEED	Set QuadSPI write speed (clock rate)	*param_ptr* - pointer to uint32_t
QuadSPI_IOCTL_SET_FLASH_ADDR	Set external flash address with memory mapped address.	*param_ptr* - pointer to uint32_t
QuadSPI_IOCTL_SET_SINGLE_IO	Set 1x pad IO mode	none (NULL)

Table continues on the next page...

Table 11-2. Supported IOCTRL Commands (continued)

Command	Description	Parameters
QuadSPI_IOCTL_SET_DUAL_IO	Set 2x pad IO mode	none (NULL)
QuadSPI_IOCTL_SET_QUAD_IO	Set 4x pad IO mode	none (NULL)
QuadSPI_IOCTL_GET_MEM_BASE_ADDR	Get base address of flash memory	*param_ptr* - pointer to uint32_t
QuadSPI_IOCTL_INVALID_BUFFER	Invalid AHB buffer	*param_ptr* - pointer to QuadSPI_MEM_BLOCK_STRUCT
QuadSPI_IOCTL_GET_IO_MODE	Get current IO mode	*param_ptr* - pointer to uint32_t
QuadSPI_IOCTL_GET_CLK_MODE	Get current clock mode	*param_ptr* - pointer to uint32_t
QuadSPI_IOCTL_GET_FLASH_INFO	Get Flash information	*param_ptr* - pointer to QuadSPI_FLASH_INFO_STRUCT_PTR
QuadSPI_IOCTL_GET_MEM_TOTAL_LEN	Get total size of Flash memory	*param_ptr* - pointer to uint32_t
QuadSPI_IOCTL_GET_PARA_MODE	Check AHB access parallel mode	*param_ptr* - pointer to bool
QuadSPI_IOCTL_GET_IP_PARA_MODE	Check IP access parallel mode	*param_ptr* - pointer to bool
QuadSPI_IOCTL_SET_IP_PARA_MODE	Set IP access parallel mode	*param_ptr* - pointer to bool

11.5.3 Access External Flash

Low level driver provides a specific method to access the external flash. Different QuadSPI flash devices might need different command sequences. To make the driver more generic and to enable the support of the external flash device, the QuadSPI low level driver for Vybrid provides basic functions to access the external flash without checking the meaning of commands. The SDK or an example manages the specified commands for the external flash device.

11.6 Send Command Structure

The command format of QuadSPI is defined as shown in the following figure. QuadSPI driver supports sending commands sequentially until it encounters the STOP instruction. Each command sequence can have a maximum of 8 commands. All commands must end with the STOP instruction (QuadSPI_LOOKUP_STOP).

Figure 11-1. QuadSPI Command Format

mqx/source/io/qspi/qspi_quadspi.h defines 6-bits *instruction*. To find the definition of the 6 bits instruction, see the QuadSPI chapter of the Vybrid reference manual.

mqx/ examples/qspi/qspi_memory.h defines 8-bits *operand* for Spansion S25FL128S and S25FL256S. The definition of an operand is defined by a specific flash device. See the datasheet of the external flash device.

Note

All commands should follow the expected sequence of the specified external QuadSPI flash device.

11.7 *fwrite* Usage

fwrite provides two functionalities: sending command to the external flash and programing the external flash. The format of *fwrite* is:

fwrite: Commands + Write Buffer Address + Write/Read Bytes (**optional**)

Sending a command to the flash:

ioctl(qspifd, QuadSPI_IOCTL_SET_FLASH_ADDR, &addr); (**optional**)

fwrite: Commands + Write Buffer Address (NULL)

Programing data to the flash:

ioctl(qspifd, QuadSPI_IOCTL_SET_FLASH_ADDR, &addr);

fwrite: Commands + Write Buffer Address + Write Bytes

Sending a read command to the flash

ioctl(qspifd, QuadSPI_IOCTL_SET_FLASH_ADDR, &addr);

fwrite: Commands + Write Buffer Address(NULL) + Read Bytes

fwrite operation is not synchronous. The calling task returns when it finishes sending commands to the flash.

11.8 *fread* Usage

The read sequence is set through a sent command to the external flash by *fwrite*. *fread* must follow after sending the command. *fread* is only used for reading data from the RX buffers. If the RX buffer is empty, *fread* will return an error.

fread operation is synchronous. The calling task is always blocked until the read operation is complete or an error occurs. *fread* returns after a given number of bytes was read.

11.9 Example

The following examples describe the QSPI driver functionality.

11.9.1 Example: Send Command to Flash

To send a command to the external flash, please provide 4 bytes **NULL** address after the commands. The following function is an example to show how to send a write-enable command to the external flash. All variable and macro definitions can be found in the qspi example (*mqx/examples/qspi)*.

```
    buffer[0] = QuadSPI_WRITE_EN & 0xFF;
    buffer[1] = (QuadSPI_WRITE_EN >> 8) & 0xFF;
    buffer[2] = QuadSPI_LOOKUP_STOP & 0xFF;
    buffer[3] = (QuadSPI_LOOKUP_STOP >> 8) & 0xFF;
    /* Write instruction */
    result = fwrite (buffer, 1, 4 + QuadSPI_ADDR_BYTES, qspifd);
    if (result < 0) {
        printf ("ERROR\n");
        return;
}
/* Wait till the flash write enable is set */
    memory_wait_for_write_en(qspifd);
```

fwrite operation is not synchronous. The calling task returns when it finishes sending commands to the flash. To determine if the command is accepted by the external flash, check the status of the external flash. The function `memory_wait_for_write_en(qspifd)` in this example is used to check if the flash accepts the write enable command. Please see Chapter Example: Read Status from External Flash for more details.

11.9.2 Example: Program Flash

To program data to the external flash, follow these steps:

1. Send write enable command to the external flash to make sure that the flash device can accept the program commands.
2. Specify the programming location for the flash address.
3. Send a program command, followed by the start address and the size of data array which needs to be programmed to the external flash.

The following function is an example which shows how to page the program data to the external flash. All variable and macro definitions can be found in the qspi example in (*mqx/examples/qspi*).

```
/* Set flash to accept write command */ memory_set_write_en(qspifd, TRUE);
/* Set flash start address to be programmed */
ioctl(qspifd, QuadSPI_IOCTL_SET_FLASH_ADDR, &addr);
/* Write instruction, address and data to buffer */
buffer[0] = QuadSPI_SET_PAGE_WR & 0xFF;
buffer[1] = (QuadSPI_SET_PAGE_WR >> 8) & 0xFF;
buffer[2] = QuadSPI_SET_ADDR & 0xFF;
buffer[3] = (QuadSPI_SET_ADDR >> 8)& 0xFF;
buffer[4] = QuadSPI_WRITE_DATA(page_size) & 0xFF;
buffer[5] = (QuadSPI_WRITE_DATA(page_size) >> 8) & 0xFF;
buffer[6] = QuadSPI_LOOKUP_STOP & 0xFF;
buffer[7] = (QuadSPI_LOOKUP_STOP >> 8) & 0xFF;
addr_to_data_buf((uint32_t) src_ptr, &(buffer[8]));
result = fwrite (buffer, 1, 8 + QuadSPI_ADDR_BYTES + write_size, qspifd);
if (result < 0) {
printf ("ERROR\n");
      return -1;
}
/* Wait till the flash is not busy at program */
memory_wait_for_not_busy(qspifd);
```

11.9.3 Example: Read from External Flash

To read the data from the external flash, follow these steps:

1. Specify the programming location of the flash address.
2. Send the specified read command to the external flash device by a required sequence such as SDR single-pad read mode or DDR single-pad fast read mode.
3. Receive and read data from the external flash device by using *fread*.

The following function is an example which shows how to read data from the external flash by DDR single-pad fast read mode. All variable and macro definitions can be found in the qspi example in (*mqx/examples/qspi*).

```
/* Set flash start address to read from */
ioctl(qspifd, QuadSPI_IOCTL_SET_FLASH_ADDR, &addr);
/* Read instruction, address */
buffer[0] = QuadSPI_SET_DDR_FAST_RD & 0xFF;
buffer[1] = (QuadSPI_SET_DDR_FAST_RD >> 8) & 0xFF;
buffer[2] = QuadSPI_SET_DDR_ADDR & 0xFF;
```

```
buffer[3] = (QuadSPI_SET_DDR_ADDR >> 8) & 0xFF;
buffer[4] = QuadSPI_SET_MODE_DDR(0xFF) & 0xFF;
buffer[5] = (QuadSPI_SET_MODE_DDR(0xFF) >> 8) & 0xFF;
buffer[6] = QuadSPI_DUMMY_DATA(0x02) & 0xFF;
buffer[7] = (QuadSPI_DUMMY_DATA(0x02) >> 8) & 0xFF;
buffer[8] = QuadSPI_DDR_READ_DATA(read_size) & 0xFF;
buffer[9] = (QuadSPI_DDR_READ_DATA(read_size) >> 8) & 0xFF;
buffer[10] = QuadSPI_LOOKUP_STOP & 0xFF;
buffer[11] = (QuadSPI_LOOKUP_STOP >> 8) & 0xFF;
/* Write instruction and address */
result = fwrite (buffer, 1, 12 + QuadSPI_ADDR_BYTES + read_size, qspifd);
if (result < 0) {
/* Stop transfer */
      printf("ERROR (tx)\n");
      return -1;
}
/* Read data from memory */
result = fread (src_ptr, 1, read_size, qspifd);
if (result < 0) {
printf ("ERROR (rx)\n");
return -1;
}
```

11.9.4 Example: Read Status from External Flash

The external flash device provides status registers to indicate the status of the flash. The status can be used to assess whether the flash has completed an operation, for example, whether the flash is busy for programming. To read the status from the external flash, follow these steps:

1. Send a specified read status command to the external flash device by a requested sequence.
2. Receive and read status data from the external flash device by using *fread*.

The following function is an example which shows how to read the status register-1 from the external flash. All variable and macro definitions can be found in the QSPI example in (*mqx/examples/qspi*).

```
buffer[0] = QuadSPI_READ_STATUS1 & 0xFF;
buffer[1] = (QuadSPI_READ_STATUS1 >> 8) & 0xFF;
/* 1 byte status-1*/
buffer[2] = QuadSPI_READ_DATA(1) & 0xFF;
buffer[3] = (QuadSPI_READ_DATA(1) >> 8) & 0xFF;
buffer[4] = QuadSPI_LOOKUP_STOP & 0xFF;
buffer[5] = (QuadSPI_LOOKUP_STOP >> 8) & 0xFF;
/* Write instruction */
result = fwrite(buffer, 1, 6 + QuadSPI_ADDR_BYTES + 1, qspifd);
if(result < 0) {
    printf("ERROR (tx)\n");
    return FALSE;
}
/* Read memory status: 1byte */
result = fread (status, 1, 1, qspifd);
```

11.9.5 Example: Erase External Flash Chip

To erase the external flash chip, send the erase command to the external flash. The following function is an example which shows how to erase the external flash chip. All macro definitions could be found in the qspi example in (*mqx/examples/qspi*).

```
/* Enable flash memory write */
memory_set_write_en(qspifd, TRUE);
/* Send erase command */
buffer[0] = QuadSPI_CHIP_ERASE & 0xFF;
buffer[1] = (QuadSPI_CHIP_ERASE >> 8) & 0xFF;
buffer[2] = QuadSPI_LOOKUP_STOP & 0xFF;
buffer[3] = (QuadSPI_LOOKUP_STOP >> 8) & 0xFF;
/* Write instruction */
result = fwrite(buffer, 1, 4 + QuadSPI_ADDR_BYTES, qspifd);
/* Wait till the flash is not busy at program */
memory_wait_for_not_busy(qspifd);
```

11.10 Error Codes

Table 11-3. Error Codes

Error Code	Descriptoin
QSPI_OK	Success.
QuadSPI_INVAID_PARAMETER	Invalid parameter.

Chapter 12
I2C Driver

12.1 Overview

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

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

The I^2C interrupt-driven I/O driver, which was included in version 4.1.0 and earlier, supported the asynchronous (non-blocking) access only. In the version 4.1.1 the Kinetis driver variant was updated to support synchronous blocking access to the I/O read() and write() functionality. When the application calls the _io_read() function, the function returns when the driver reads the specified number of bytes. The calling task is blocked until the complete buffer is read.

- I^2C polled I/O - polled driver which provides synchronous blocking functionality.

12.2 Source Code Location

Table 12-1. Source code location

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

12.3 Header Files

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

Table 12-2. Header files

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

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

12.4 Installing Drivers

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

Table 12-3. Function names

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

12.4.1 Initialization Records

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

Synopsis for kinetis family, mcf51jf and mcf51qm

```
#include <i2c_ki2c.h>
typedef struct ki2c_init_struct
{
  uint8_t        CHANNEL;
  uint8_t        MODE;
  uint8_t        STATION_ADDRESS;
  uint32_t       BAUD_RATE;
#if !(BSP_TWRMCF51FD || BSP_TWRMCF51JF || BSP_TWRMCF51QM)
  _int_level     LEVEL;
  _int_priority  SUBLEVEL;
#endif
  uint32_t       TX_BUFFER_SIZE;
  uint32_t       RX_BUFFER_SIZE;
}    KI2C_INIT_STRUCT, * KI2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE).

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

BAUD_RATE - Desired baud rate.

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

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

TX_BUFFER_SIZE - Tx buffer size (interrupt driven only).

RX_BUFFER_SIZE - Rx buffer size (interrupt driven only).

Synopsis for mcf51XX family (except of mcf51jf and mcf51qm)

```
#include <i2c_mcf51xx.h>
typedef struct mcf51xx_i2c_init_struct
{
  uint8_t  CHANNEL;
  uint8_t  MODE;
  uint8_t  STATION_ADDRESS;
  uint32_t BAUD_RATE;
  uint32_t TX_BUFFER_SIZE;
  uint32_t RX_BUFFER_SIZE;
} MCF51XX_I2C_INIT_STRUCT, * MCF51XX_I2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE).

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

BAUD_RATE - Desired baud rate.

TX_BUFFER_SIZE - Tx buffer size (interrupt driven only).

RX_BUFFER_SIZE - Rx buffer size (interrupt driven only).

Synopsis for mcf52XX

```
#include <i2c_mcf52xx.h>
typedef struct mcf52xx_i2c_init_struct
{
  uint8_t       CHANNEL;
  uint8_t       MODE;
  uint8_t       STATION_ADDRESS;
  uint32_t      BAUD_RATE;
  _int_level    LEVEL;
  _int_priority SUBLEVEL;
  uint32_t      TX_BUFFER_SIZE;
  uint32_t      RX_BUFFER_SIZE;
} MCF52XX_I2C_INIT_STRUCT, * MCF52XX_I2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE).

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

BAUD_RATE - Desired baud rate.

LEVEL - Interrupt level to use if interrupt driven.

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

TX_BUFFER_SIZE - Tx buffer size (interrupt driven only).

RX_BUFFER_SIZE - Rx buffer size (interrupt driven only).

Synopsis for mcf53XX and mcf54XX (example for mcf53XX)

```
#include <i2c_mcf53xx.h>
typedef struct mcf53xx_i2c_init_struct
{
  uint8_t     CHANNEL;
  uint8_t     MODE;
  uint8_t     STATION_ADDRESS;
  uint32_t    BAUD_RATE;
  _int_level  LEVEL;
  uint32_t    TX_BUFFER_SIZE;
  uint32_t    RX_BUFFER_SIZE;
} MCF53XX_I2C_INIT_STRUCT, * MCF53XX_I2C_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2C channel to initialize.

MODE - Default operating mode (I2C_MODE_MASTER or I2C_MODE_SLAVE).

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

BAUD_RATE - Desired baud rate.

LEVEL - Interrupt level to use if interrupt driven.

TX_BUFFER_SIZE - Tx buffer size (interrupt driven only).

RX_BUFFER_SIZE - Rx buffer size (interrupt driven only).

Example

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

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

12.5 Driver Services

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

Table 12-4. Driver 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()

12.6 I/O Control Commands

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

Table 12-5. I/O control commands

Command	Description	Parameters
IO_IOCTL_I2C_SET_BAUD	Sets the baud rate.	*param_ptr* - pointer to uint32_t
IO_IOCTL_I2C_GET_BAUD	Gets the baud rate.	*param_ptr* - pointer to uint32_t
IO_IOCTL_I2C_SET_MASTER_MODE	Sets device to the I^2C master mode.	none (NULL)
IO_IOCTL_I2C_SET_SLAVE_MODE	Sets device to the I^2C slave mode	none (NULL)
IO_IOCTL_I2C_GET_MODE	Gets the mode previously set.	*param_ptr* - pointer to uint32_t (I2C_MODE_MASTER or I2C_MODE_SLAVE)
IO_IOCTL_I2C_SET_STATION_ADDRESS	Sets the device's I^2C slave address.	*param_ptr* - pointer to uint32_t
IO_IOCTL_I2C_GET_STATION_ADDRESS	Gets the device's I^2C slave address.	*param_ptr* - pointer to uint32_t
IO_IOCTL_I2C_SET_DESTINATION_ADDRESS	Sets the address of the called device (master only).	*param_ptr* - pointer to uint32_t
IO_IOCTL_I2C_GET_DESTINATION_ADDRESS	Gets the address of the called device (master only).	*param_ptr* - pointer to uint32_t
IO_IOCTL_I2C_SET_RX_REQUEST	Sets a number of bytes in advance to read before stop.	*param_ptr* - pointer to uint32_t
IO_IOCTL_I2C_REPEATED_START	Initiates I^2C repeated start condition (master only).	none (NULL)
IO_IOCTL_I2C_STOP	Generates I^2C stop condition (master only).	none (NULL)
IO_IOCTL_I2C_GET_STATE	Gets the actual state of transmission.	*param_ptr* - pointer to uint32_t (see I2C state enum type in i2c.h)
IO_IOCTL_I2C_GET_STATISTICS	Gets the communication statistics (structure defined in *i2c.h*.)	*param_ptr* - pointer to I2C_STATISTICS_STRUCT
IO_IOCTL_I2C_CLEAR_STATISTICS	Clears the communication statistics.	none (NULL)
IO_IOCTL_I2C_DISABLE_DEVICE	Disables I^2C device.	none (NULL)
IO_IOCTL_I2C_ENABLE_DEVICE	Enables I^2C device.	none (NULL)
IO_IOCTL_FLUSH_OUTPUT	Flushes the output buffer, waits for the transfer to finish.	none (NULL)
IO_IOCTL_I2C_GET_BUS_AVAILABILITY	Gets the actual bus state (idle/busy).	*param_ptr* - pointer to uint32_t (I2C_BUS_BUSY or I2C_BUS_IDLE)

12.7 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*.

Table 12-6. Device state values

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

12.8 Device Modes

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

Table 12-7. Device mode values

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.

12.9 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*.

Table 12-8. Bus states

Bus State	Description
I2C_BUS_IDLE	Stop condition occurred. No i2c transmission on the bus.

Table continues on the next page...

Table 12-8. Bus states (continued)

Bus State	Description
I2C_BUS_BUSY	Start/Repeated started detected. Transmission in progress.

12.10 Error Codes

No additional error codes are generated.

Table 12-9. Error codes

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 13
FlashX Driver

13.1 Overview

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

13.2 Source Code Location

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

13.3 Header Files

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

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

13.4 Hardware Supported

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

13.5 Driver Services

Flash drivers provide the following full set of services.

Table 13-1. Flash driver 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()

13.6 Installing Drivers

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

13.7 Installing and Uninstalling Flash Devices

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

This function initializes the generic driver.

13.7.1 _io_flashx_install

Synopsis

```
_max_uint _io_flashx_install(char *id, FLASHX_INIT_STRUCT *init_ptr)
```

Parameters

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

13.7.2 _io_flashx_uninstall

Synopsis

```
_max_uint _io_flashx_uinstall()
```

13.7.3 FLASHX_INIT_STRUCT

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

Parameters

- BASE_ADDR [IN] — Base address of the device.
- HW_BLOCK [IN] — Array of HW blocks describing the organization of Flash memory.
- FILE_BLOCK [IN] — Array of BSP predefined files that can be opened with the Flash driver.
- DEVICE_IF [IN] — Array of device interface which includes functions that map functionality.
- WIDTH [IN] — The bus data lines used for external devices.
  - 1 (accessed by bytes)
  - 2 (accessed by words)

- 4 (accessed as long words)
- 8 (accessed as double longs)

- WRITE_VERIFY [IN] — If true, a comparison of the original data and the flash write is made.
- DEVICE_SPECIFIC_INIT [IN] — If required by the low level driver, user can pass information from the BSP.

13.7.4 FLASHX_BLOCK_INFO_STRUCT

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

Synopsis

```
struct flashx_block_info_struct {
  _mqx_uint        NUM_SECTORS;
  _mem_size        START_ADDR;
  _mem_size        SECT_SIZE;
  uint32_t         SPECIAL_TAG;
} FLASHX_BLOCK_INFO_STRUCT, * FLASHX_BLOCK_INFO_STRUCT;
```

Parameters

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

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

```
#define AT49BV1614A_SECTOR_SIZE_1 (0x2000)
#define AT49BV1614A_SECTOR_SIZE_2 (0x10000)
#define AT49BV1614A_NUM_SECTORS_1 (8)
#define AT49BV1614A_NUM_SECTORS_2 (31)
FLASHX_BLOCK_INFO_STRUCT _at49bv1614a_block_map_16bit[] =  {
  { AT49BV1614A_NUM_SECTORS_1, 0, AT49BV1614A_SECTOR_SIZE_1, 0},                          /*
8x8KB */
  { AT49BV1614A_NUM_SECTORS_2, 0x10000, AT49BV1614A_SECTOR_SIZE_2, 0},/* 31x64KB */
  { 0, 0, 0, 0 } /* zero block terminates the array */
};
```

13.7.5 FLASHX_FILE_BLOCK

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

Synopsis

```
typedef struct flashx_file_block
{
    char *const   FILENAME;
    _mem_size     START_ADDR;
    _mem_size     END_ADDR;
} FLASHX_FILE_BLOCK, * FLASHX_FILE_BLOCK_PTR;
```

Parameters

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

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

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

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

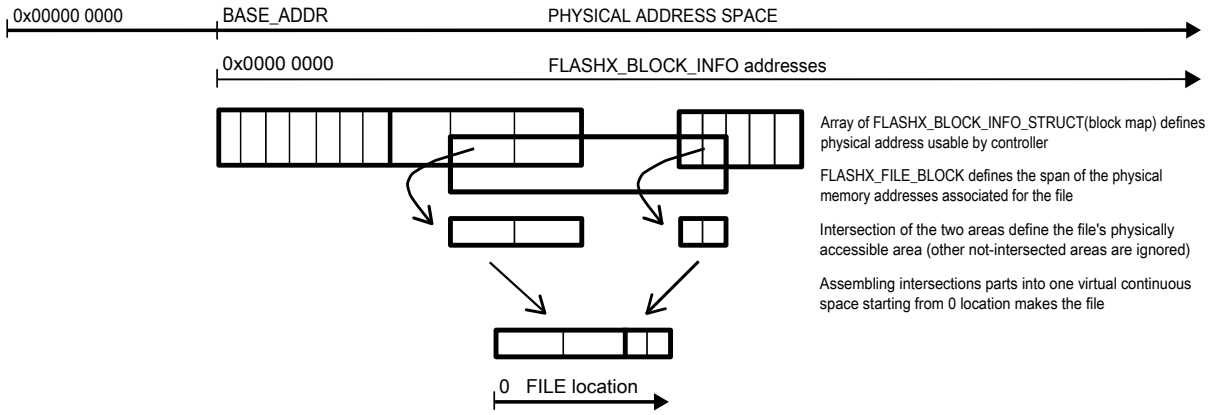

Figure 13-1. File address

13.8 I/O Control Commands

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

Table 13-2. I/O control commands

Command	Description	Parameters
FLASH_IOCTL_GET_BASE_ADDRESS	Returns the base address of the flash memory.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_NUM_SECTORS	Returns the number of sectors in the flash file.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_SECTOR_BASE	Returns the start address of the current sector - sector at current file location.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_SECTOR_SIZE	Returns the size of the current sector - sector at current file location.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_WIDTH	Returns the width of the flash device.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_BLOCK_GROUPS	Returns the number of blocks in the device block map.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_GET_BLOCK_MAP	Returns the address of the device block map description defined by an array of FLASHX_BLOCK_INFO_STRUCT.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_FLUSH_BUFFER	Writes out all cached sectors if there is valid data in the cache.	none (NULL)
FLASH_IOCTL_ENABLE_BUFFERING	Enables write-back caching of the single Flash sector.	none (NULL)

Table continues on the next page...

Table 13-2. I/O control commands (continued)

Command	Description	Parameters
	This **ioctl** can only be enabled if FLASH_IOCTL_ENABLE_SECTOR _CACHE is enabled.	
FLASH_IOCTL_DISABLE_BUFFERING	Disables write back cache.	none (NULL)
FLASH_IOCTL_ERASE_SECTOR	Erases the sector at a current file location.	none (NULL)
FLASH_IOCTL_ERASE_CHIP	Erases the entire flash device.	none (NULL)
FLASH_IOCTL_ENABLE_SECTOR_CACHE	Enables allocating the sector buffer. This affects write-back caching. Also, it restricts some driver functionality. See FLASH_IOCTL_DISABLE_SECTOR _CACHE).	none (NULL)
FLASH_IOCTL_DISABLE_SECTOR_CACHE	Disables allocating the sector in the memory. Intention of this feature is RAM saving, the sector size could be large enough to decrease performance of the application, but it restricts driver functionality. Sector allocation is needed in the following cases: Partial sector overwrite when the destination area is not blank. In these cases the sector allocation is **not** required: - Full sector write. - Partial sector overwrite when the destination area is blank. Disabling sector cache also rules out write-back caching. See FLASH_IOCTL_ENABLE_BUFFERI NG.	none (NULL)
IO_IOCTL_GET_NUM_SECTORS	Returns the number of sectors for MFS device. The default MSF_SECTOR_SIZE is 512 bytes.	*param_ptr* - pointer to 32b variable
IO_IOCTL_DEVICE_IDENTIFY	Returns to upper layer, what kind of device is it. It is a physical flash device, capable of being erased, read, and written. Flash devices are not interrupt driven, so IO_DEV_ATTR_POLL is included. Used in MFS driver.	*param_ptr* - pointer to block identification array, required by MFS
IO_IOCTL_GET_BLOCK_SIZE	Returns the fixed MFS sector size which is usually 512.	*param_ptr* - pointer to 32b variable
FLASH_IOCTL_SWAP_FLASH_AND_RESET	Swaps the flash memory blocks. Works only with the dual flash memory controllers.	none (NULL)

Table continues on the next page...

Table 13-2. I/O control commands (continued)

Command	Description	Parameters
FLASH_IOCTL_WRITE_ERASE_CMD_FROM_FLASH_ENABLE	Sets up to run the low level flash write and erase routines from internal flash memory. Supported only on the dual flash memory controllers.	none (NULL)
FLASH_IOCTL_WRITE_ERASE_CMD_FROM_FLASH_DISABLE	Sets up to run the low level flash write and erase routines from RAM. Supported only on the dual flash memory controllers.	none (NULL)

The following table lists the FlexNVM specific IOCTL commands.

Table 13-3. 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 uint8_t - 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 uint32_t - EEReady flag value: 0x1 - ready

13.9 Data Types Used with the FlexNVM

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

13.9.1 FLEXNVM_READ_RSRC_STRUCT

Synopsis:

```
typedef struct {
  uint32_t ADDR;
  uint8_t  RSRC_CODE;
  uint32_t RD_DATA;
} FLEXNVM_READ_RSRC_STRUCT;
```

Parameters:

ADDR - flash address.

RSRC_CODE - resource selector.

RD_DATA - readed resources data.

13.9.2 FLEXNVM_PROG_PART_STRUCT

Synopsis:

```
typedef struct {
  uint8_t EE_DATA_SIZE_CODE;
  uint8_t FLEXNVM_PART_CODE;
} FLEXNVM_PROG_PART_STRUCT;
```

Parameters:

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

Configuration values for EE_SPLIT are:

- FLEXNVM_EE_SPLIT_1_7
- FLEXNVM_EE_SPLIT_1_3
- FLEXNVM_EE_SPLIT_1_1

Configuration values for EE_SIZE are:

- FLEXNVM_EE_SIZE_4096
- FLEXNVM_EE_SIZE_2048
- FLEXNVM_EE_SIZE_1024
- FLEXNVM_EE_SIZE_512
- FLEXNVM_EE_SIZE_256

- FLEXNVM_EE_SIZE_128
- FLEXNVM_EE_SIZE_64
- FLEXNVM_EE_SIZE_32
- FLEXNVM_EE_SIZE_0

FLEXNVM_PART_CODE - FlexNVM partition code. Possible values are:

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

13.10 Error Codes

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

Chapter 14
SD Card Driver

14.1 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 -.

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.

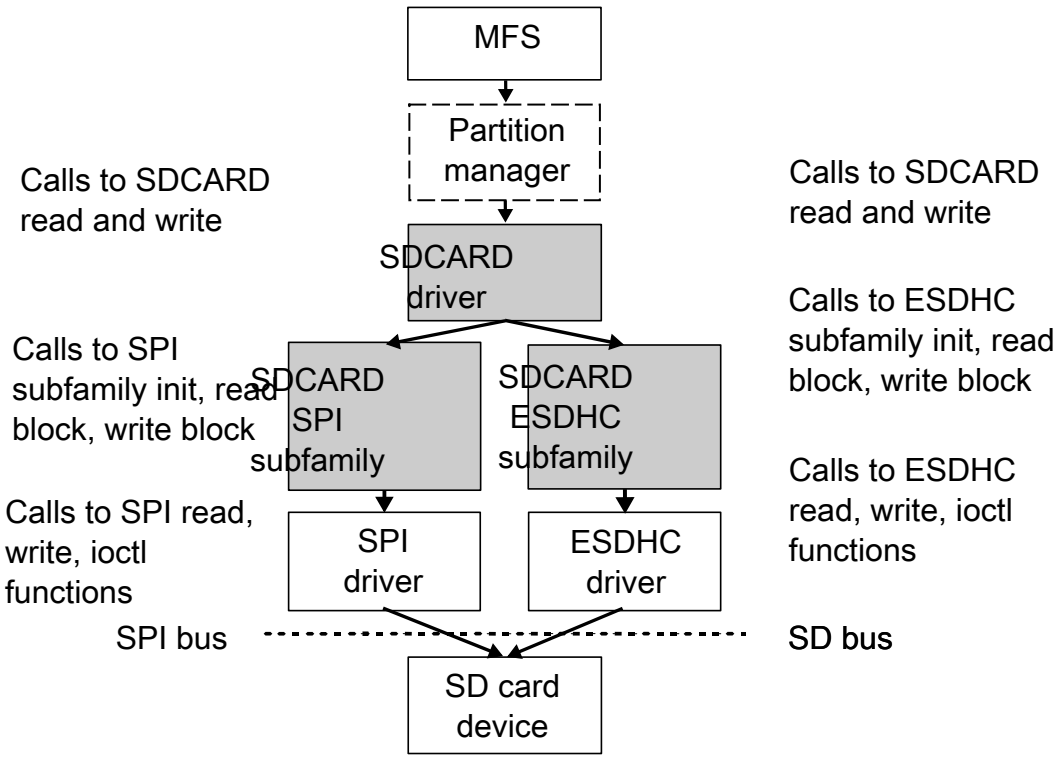

Figure 14-1. SD Card driver stack

14.2 Source Code Location

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

14.3 Header Files

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

14.4 Installing Driver

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

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

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

```
_mqx_int _io_sdcard_install
   (
      /* [IN] A string that identifies the device for fopen */
      char                  *identifier,
      /* [IN] SD card initialization parameters */
      SDCARD_INIT_STRUCT_PTR init,
      /* [IN] Already opened communication descriptor */
      FILE_PTR               com_device
   )
```

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

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

14.4.1 Initialization Record

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

Synopsis

```
#include <sdcard.h>
typedef struct sdcard_init_struct
{
bool (_CODE_PTR_ INIT_FUNC)(MQX_FILE_PTR);
bool (_CODE_PTR_ READ_FUNC)(MQX_FILE_PTR, unsigned char*, uint32_t);
bool (_CODE_PTR_ WRITE_FUNC)(MQX_FILE_PTR, unsigned char*, uint32_t);
```

```
uint32_t SIGNALS;
} SDCARD_INIT_STRUCT, * SDCARD_INIT_STRUCT_PTR;
```

Parameters for SPI interface

INIT_FUNC - initialization function, set to `_io_sdcard_spi_init`.

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

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

SIGNALS - determines SPI chip select for SDCARD communication.

Parameters for SDHC interface

INIT_FUNC - initialization function, set to `_io_sdcard_sdhc_init`.

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

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

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

Parameters for ESDHC interface

INIT_FUNC - initialization function, set to `_io_sdcard_esdhc_init`.

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

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

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

Example

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

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

14.4.2 Driver Services

The SD Card device driver provides these services:

Table 14-1. Driver services

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

14.5 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*.

Table 14-2. I/O control commands

Command	Description	Parameters
IO_IOCTL_GET_BLOCK_SIZE	Returns the size of block in bytes. This ioctl command is mandatory for using a device with MFS.	*param_ptr* - pointer to uint32_t
IO_IOCTL_GET_NUM_SECTORS	Returns number of blocks available in the SD card.This ioctl command is mandatory for using a device with MFS.	*param_ptr* - pointer to _mem_size
IO_IOCTL_DEVICE_IDENTIFY	Returns flags describing SD card capabilities.This ioctl command is mandatory for using device with MFS.	*param_ptr* - pointer to uint32_t
IO_IOCTL_SDCARD_GET_CID	Retrieves 16-byte card identification (CID) of the card. The parameter to this IOCTL call is used as pointer to caller allocated buffer used to return the CID.	*param_ptr* - pointer to 16-byte buffer the CID is copied to

14.6 Example

See example provided with the MQX RTOS installation located in: mfs/examples/sdcard directory.

Chapter 15
RTC Driver

15.1 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).

15.2 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.

15.3 Header Files

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

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

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

15.4 API Function Reference - RTC Module Related Functions

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

15.4.1 _rtc_init()

This function (re)initializes the RTC module.

Synopsis

```
uint32_t _rtc_init(void* param)
```

Parameters

Return Value

- MQX_OK (success)
- Other value if unsuccessful

Example

The following example shows how to initialize the RTC module.

```
_rtc_init(NULL);
```

15.4.2 _rtc_isr()

This is the interrupt service routine for the RTC module.

Synopsis

```
void _rtc_isr(void *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

15.4.3 _rtc_callback_reg()

This function installs the ISR for the RTC module.

Synopsis

```
int32_t _rtc_int_install(INT_ISR_FPTR func, void* data)
```

Parameters

func [in] — pointer to user ISR code.

data[in] — pointer to user ISR data.

Description

This function installs the given user callback for RTC module. The user callback is called when RTC alarm occurred.

Return Value

- MQX_OK (Success)
- Other value if not successful

Example

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

```
printf ("Installing alarm callback... ");
if (MQX_OK != _rtc_callback_reg (my_callback, NULL))
{
printf ("Error!\n");
}
```

15.4.4 _rtc_set_time()

This function sets the RTC time.

Synopsis

```
int32_t  _rtc_set_time(uint32_t time)
```

Parameters

time [in] —the number of seconds to be set as RTC time value.

Description

This function sets the RTC time according to the given number of seconds.

Return Value

- MQX_OK (Success)
- Other value if not successful

Example

The following example shows how to set RTC time to 27-9-2013 9:47:15.

```
DATE_STRUCT     date_time;
TIME_STRUCT     mqx_time;
uint32_t        rtc_time;
date_time.YEAR  = 2013;
date_time.MONTH = 9;
date_time.DAY   = 27;
date_time.HOUR  = 9;
date_time.MINUTE= 47;
date_time.SECOND= 15;
date_time.MILLISEC = 0;
_time_from_date(&date_time, &mqx_time);
rtc_time = mqx_time.SECONDS;
_rtc_set_time(rtc_time);
```

15.4.5 _rtc_get_time()

This function returns the actual RTC time.

Synopsis

```
int32_t _rtc_get_time(uint32_t* 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

- MQX_OK (Success)
- Other value if not successful

15.4.6 rtc_set_alarm()

This function sets the RTC alarm.

Synopsis

```
int32_t _rtc_set_alarm(uint32_t time, uint32_t     period)
```

Parameters

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

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

Description

This function sets the RTC alarm according to the given alarm time and alarm period.

Return Value

- MQX_OK (Success)
- RTC_INVALID_TIME (Not Success)

Example

The following example shows how to set the RTC alarm time to 1.1.2014, 12:30:00 with period time being 4 seconds.

```
DATE_STRUCT     date_time;
TIME_STRUCT     mqx_time;
uint32_t        alarm_time;
date_time.YEAR  = 2014;
date_time.MONTH = 1;
date_time.DAY   = 1;
date_time.HOUR  = 12;
date_time.MINUTE= 30;
date_time.SECOND= 0;
date_time.MILLISEC = 0;
_time_from_date(&date_time, &mqx_time);
alarm_time = mqx_time.SECONDS;
_rtc_set_alarm(alarm_time, 4);
```

15.4.7 _rtc_get_alarm()

This function returns the RTC alarm time.

Synopsis

```
int32_t _rtc_get_alarm(uint32_t* 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

- MQX_OK (Success)

15.5 API Function Reference - IRTC Module Specific Functions

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

15.5.1 _rtc_lock()

This function locks RTC registers.

Synopsis

```
void _rtc_lock(void)
```

Parameters

None

Description

This function locks RTC registers.

Return Value

- None

15.5.2 _rtc_unlock()

This function unlocks RTC registers.

Synopsis

```
int32_t rtc_unlock(void)
```

Parameters

None

Description

This function unlocks RTC registers.

Return Value

- MQX_OK (Success)
- Other value if not successful

15.5.3 _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

15.5.4 _rtc_get_upcounter()

This function returns value of the up-counter register.

Synopsis

```
uint32_t _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

15.5.5 _rtc_write_to_standby_ram()

This function writes to the stand-by RAM.

Synopsis

```
int32_t _rtc_write_to_standby_ram(
        uint32_t dst_address,
        uint8_t *src_ptr,
        uint32_t 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

15.5.6 _rtc_read_from_standby_ram()

This function reads from the standby RAM.

```
int32_t _rtc_read_from_standby_ram(
        uint32_t src_address,
        uint8_t *dst_ptr,
        uint32_t 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

15.6 Data Types Used by the RTC Driver API

The following data types are used within the RTC driver.

15.6.1 RTC TIME

RTC time is stored as a 32-bit unsigned integer representing the number of seconds past the Unix epoch: midnight UTC, 1 January 1970.

15.6.2 Example

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

15.7 Error Codes

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

Chapter 16
ESDHC Driver

16.1 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.

16.2 Source Code Location

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

16.3 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.

16.4 Installing Driver

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

ESDHC device driver installation

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

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

16.4.1 Initialization Record

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

Synopsis

```
#include <esdhc.h>
typedef struct esdhc_init_struct
{
  uint32_t CHANNEL;
  uint32_t BAUD_RATE;
  uint32_t CLOCK_SPEED;
} ESDHC_INIT_STRUCT, * ESDHC_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - device number.

BAUD_RATE - desired communication baud rate.

CLOCK_SPEED - module input clock speed.

Example of ESDHC device driver initialization

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

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

16.5 Driver Services

The table below describes the ESDHC device driver services:

Table 16-1. Driver services

API	Calls	Description
`_io_fopen()`	`_esdhc_open()`	Resets the HW module. It also applies default settings (e.g., initial 400 kHz baudrate), pin assignments, sends 80 dummy clocks, and detects the presence of the card.
`_io_fclose()`	`_esdhc_close()`	Resets the HW module.
`_io_read()`	`_esdhc_read()`	Can be called only after successful data transfer command. They return after given number of bytes was transferred. After the whole transmission, `_io_fflush()` should be called to wait for transfer complete flag and to check transfer errors at the host side.
`_io_write()`	`_esdhc_write()`	
`_io_ioctl()`	`_esdhc_ioctl()`	Sets up the host (card must be set up accordingly via commands over the bus). The ioctl command `IO_IOCTL_ESDHC_INIT` is called after `_io_fopen()` to determine the type of the card, to initialize it properly, and to set the baudrate requested in initialization record.

16.6 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*.

Table 16-2. I/O control commands

Command	Description	Parameters
`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.	none (NULL)
`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.	*param_ptr* - pointer to the ESDHC_COMMAND_STRUCT

Table continues on the next page...

Table 16-2. I/O control commands (continued)

Command	Description	Parameters
IO_IOCTL_ESDHC_GET_CARD	Returns type of the card detected during IO_IOCTL_ESDHC_INIT. Also detects presence of the card.	*param_ptr* - pointer to the uint32_t (see Card Types section)
IO_IOCTL_ESDHC_GET_BAUDRATE	Returns current baudrate used.	*param_ptr* - pointer to the uint32_t
IO_IOCTL_ESDHC_SET_BAUDRATE	Sets the baudrate given as parameter. Default baudrate is specified in initialization structure.	*param_ptr* - pointer to the uint32_t
IO_IOCTL_ESDHC_GET_BUS_WIDTH	Returns current bus width used at the host side.	*param_ptr* - pointer to the uint32_t (see Bus Widths section)
IO_IOCTL_ESDHC_SET_BUS_WIDTH	Sets the bus width at the host side. It should follow the successful command that sets bus width at the card. Default bus width is 1 wire.	*param_ptr* - pointer to the uint32_t (see Bus Widths section)
IO_IOCTL_FLUSH_OUTPUT	Waits for HW transfer complete flag and checks errors at the host side. It should be called after the whole data transfer.	none (NULL)

16.7 Send Command Structure

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

Note

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

```
typedef struct esdhc_command_struct
{
  uint8_t  COMMAND;
  uint32_t ARGUMENT;
  uint32_t BLOCKS;
  uint32_t BLOCKSIZE;
  uint32_t RESPONSE[4];
} ESDHC_COMMAND_STRUCT, * ESDHC_COMMAND_STRUCT_PTR;
```

Table 16-3. Command structure

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

16.7.1 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*.

Table 16-4. Commands

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

Table continues on the next page...

Table 16-4. Commands (continued)

Command	Description
ESDHC_CMD38	Erase.
ESDHC_CMD39	Fast IO.
ESDHC_CMD40	Go IRQ state.
ESDHC_CMD42	Lock/unlock.
ESDHC_CMD52	IO R/W direct.
ESDHC_CMD53	IO R/W extended.
ESDHC_CMD55	Application specific command follows.
ESDHC_CMD56	Send/receive data block for general purpose/application specific command.
ESDHC_CMD60	R/W multiple register.
ESDHC_CMD61	R/W multiple block.
ESDHC_ACMD6	Set bus width.
ESDHC_ACMD13	Send SD status (extended).
ESDHC_ACMD22	Send number of written sectors.
ESDHC_ACMD23	Set write/erase block count.
ESDHC_ACMD41	SD application specific command send OCR.
ESDHC_ACMD42	Set/clear card detection.
ESDHC_ACMD51	Send SCR.

16.8 Card Types

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

Table 16-5. Card types

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.

16.9 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*.

Table 16-6. Bus widths

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).

16.10 Error Codes

The ESDHC device driver defines the following error codes.

Table 16-7. Error codes

Error code	Description
`ESDHC_OK`	Success.
`ESDHC_ERROR_INIT_FAILED`	Error during card initialization.
`ESDHC_ERROR_COMMAND_FAILED`	Error during command execution over the bus.
`ESDHC_ERROR_COMMAND_TIMEOUT`	No response from the card to the command.
`ESDHC_ERROR_DATA_TRANSFER`	Error during data transfer detected at the host side and returned by IO_IOCTL_FLUSH_OUTPUT.
`ESDHC_ERROR_INVALID_BUS_WIDTH`	Wrong bus width detected during get/set at the host side.

16.11 Example

```
FILE_PTR esdhc_fd;
ESDHC_COMMAND_STRUCT command;
bool sdhc;
uint32_t param, rca, sector;
uint8_t buffer[512];
/* Open ESDHC driver */
esdhc_fd = fopen ("esdhc:", NULL);
if (NULL == esdhc_fd)
{
    _task_block ();
```

Example

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

/* SDHC check */
sdhc = FALSE;
param = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_GET_CARD, &param))
{
    _task_block ();
}
if ((ESDHC_CARD_SD == param) || (ESDHC_CARD_SDHC == param) || (ESDHC_CARD_SDCOMBO == param)
|| (ESDHC_CARD_SDHCCOMBO == param))
{
    if ((ESDHC_CARD_SDHC == param) || (ESDHC_CARD_SDHCCOMBO == param))
    {
        sdhc = TRUE;
    }
}
else
{
    /* Not SD memory card */
    _task_block ();
}
/* Card identify */
command.COMMAND = ESDHC_CMD2;
command.ARGUMENT = 0;
command.BLOCKS = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
    _task_block ();
}
/* Get card relative address */
command.COMMAND = ESDHC_CMD3;
command.ARGUMENT = 0;
command.BLOCKS = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
    _task_block ();
}
rca = command.RESPONSE[0] & 0xFFFF0000;
/* Select card */
command.COMMAND = ESDHC_CMD7;
command.ARGUMENT = rca;
command.BLOCKS = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
    _task_block ();
}
/* Application specific command */
command.COMMAND = ESDHC_CMD55;
command.ARGUMENT = rca;
command.BLOCKS = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
    _task_block ();
}
/* Set bus width 4 */
command.COMMAND = ESDHC_ACMD6;
command.ARGUMENT = 2;
command.BLOCKS = 0;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
    _task_block ();
}
param = ESDHC_BUS_WIDTH_4BIT;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SET_BUS_WIDTH, &param))
```

```
{
    _task_block ();
}
/* SD card data address adjustment */
sector = 0;
if (! sdhc)
{
    sector <<= 9;
}
/* Read block command */
command.COMMAND = ESDHC_CMD17;
command.ARGUMENT = sector;
command.BLOCKS = 1;
command.BLOCKSIZE = 512;
if (ESDHC_OK != ioctl (esdhc_fd, IO_IOCTL_ESDHC_SEND_COMMAND, &command))
{
    _task_block ();
}
/* Read sector 0 */
if (512 != fread (buffer, 1, 512, esdhc_fd))
{
    _task_block ();
}
/* Wait for transfer complete and check errors at host side */
if (ESDHC_OK != fflush (esdhc_fd))
{
    _task_block ();
}
/* Close driver */
fclose (esdhc_fd);
```

Chapter 17
FlexCAN Driver

17.1 Overview

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

17.2 Source Code Location

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

17.3 Header Files

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

17.4 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.

17.4.1 FLEXCAN_Softreset()

This function (re)initializes the FlexCAN module.

Synopsis

```
uint32_t FLEXCAN_Softreset(uint8_t 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 */
uint32_t result = FLEXCAN_Softreset(0);
```

17.4.2 FLEXCAN_Start()

This function puts the FlexCAN module into a working state.

Synopsis

```
uint32_t FLEXCAN_Start(uint8_t 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 */
uint32_t result = FLEXCAN_Start(0);
```

17.4.3 FLEXCAN_Get_msg_object()

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

Synopsis

```
void *FLEXCAN_Get_msg_object(
        uint8_t  dev_num,
        uint32_t 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);
```

17.4.4 FLEXCAN_Select_mode()

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

Synopsis

```
uint32_t FLEXCAN_Select_mode(
        uint8_t  dev_num,
        uint32_t 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 */
uint32_t result = FLEXCAN_Select_mode(0,FLEXCAN_NORMAL_MODE);
```

17.4.5 FLEXCAN_Select_clk()

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

Synopsis

```
uint32_t FLEXCAN_Select_clk(
        uint8_t  dev_num,
        uint32_t 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 */
uint32_t result = FLEXCAN_Select_clk(0,FLEXCAN_IPBUS_CLK);
```

17.4.6 FLEXCAN_Initialize()

This is the main setup function of the FlexCAN module.

Synopsis

```
uint32_t FLEXCAN_Initialize(
        uint8_t  dev_num,
        uint32_t bit_timing0,
        uint32_t bit_timing1,
        uint32_t frequency,
        uint32_t 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 */
uint32_t result = FLEXCAN_Initialize(0,0,0,250,FLEXCAN_IPBUS_CLK);
```

17.4.7 FLEXCAN_Initialize_mailbox()

This function sets up one FlexCAN message buffer.

Synopsis

```
uint32_t FLEXCAN_Initialize_mailbox(
        uint8_t  dev_num,
        uint32_t mailbox_number,
        uint32_t identifier,
        uint32_t data_len_code,
        uint32_t direction,
        uint32_t format,
        uint32_t int_enable)
```

Parameters

dev_num [in] - FlexCAN device number

mailbox_number [in] - FlexCAN message buffer index

identifier[in] - FlexCAN message identifier bits

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

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

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

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

Description

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

Return Value

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

Example

```
/* setup mailbox 15 to transmit standard ID 0x7FF, 8 byte data and enable particular
interrupt */
uint32_t result = FLEXCAN_Initialize_mailbox
(0,15,0x7FF,8,FLEXCAN_TX,FLEXCAN_STANDARD,FLEXCAN_ENABLE);
```

17.4.8 FLEXCAN_Request_mailbox()

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

Synopsis

```
uint32_t FLEXCAN_Request_mailbox(
        uint8_t  dev_num,
        uint32_t mailbox_number,
        uint32_t 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 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 */
uint32_t result = FLEXCAN_Request_mailbox(0,15,FLEXCAN_STANDARD);
```

17.4.9 FLEXCAN_Activate_mailbox()

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

Synopsis

```
uint32_t FLEXCAN_Activate_mailbox(
        uint8_t  dev_num,
        uint32_t mailbox_number,
        uint32_t 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 */
uint32_t result = FLEXCAN_Activate_mailbox(0,15,FLEXCAN_MESSAGE_TRANSMIT_ONCE);
```

17.4.10 FLEXCAN_Lock_mailbox()

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

Synopsis

```
uint32_t FLEXCAN_Lock_mailbox(
        uint8_t  dev_num,
        uint32_t 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 */
uint32_t result = FLEXCAN_Lock_mailbox(0,15);
```

17.4.11 FLEXCAN_Unlock_mailbox()

This function unlocks all FlexCAN message buffers.

Synopsis

```
uint32_t FLEXCAN_Unlock_mailbox(uint8_t 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 */
uint32_t result = FLEXCAN_Unlock_mailbox(0);
```

17.4.12 FLEXCAN_Set_global_extmask()

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

Synopsis

```
uint32_t FLEXCAN_Set_global_extmask(
        uint8_t  dev_num,
        uint32_t 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 */
uint32_t result = FLEXCAN_Set_global_extmask(0,0x1FFFFFFE);
```

17.4.13 FLEXCAN_Set_buf14_extmask()

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

Synopsis

```
uint32_t FLEXCAN_Set_buf14_extmask(
        uint8_t  dev_num,
        uint32_t 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 */
uint32_t result = FLEXCAN_Set_buf14_extmask(0,0x1FFFFFFE);
```

17.4.14 FLEXCAN_Set_buf15_extmask()

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

Synopsis

```
uint32_t FLEXCAN_Set_buf15_extmask(
        uint8_t  dev_num,
        uint32_t 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 */
uint32_t result = FLEXCAN_Set_buf15_extmask(0,0x1FFFFFFE);
```

17.4.15 FLEXCAN_Set_global_stdmask()

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

Synopsis

```
uint32_t FLEXCAN_Set_global_stdmask(
        uint8_t  dev_num,
        uint32_t 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 */
uint32_t result = FLEXCAN_Set_global_stdtmask(0,0x7FE);
```

17.4.16 FLEXCAN_Set_buf14_stdmask()

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

Synopsis

```
uint32_t FLEXCAN_Set_buf14_stdmask(
        uint8_t  dev_num,
        uint32_t 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 */
uint32_t result = FLEXCAN_Set_buf14_stdmask(0,0x7FE);
```

17.4.17 FLEXCAN_Set_buf15_stdmask()

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

Synopsis

```
uint32_t FLEXCAN_Set_buf15_stdmask(
        uint8_t  dev_num,
        uint32_t 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 */
uint32_t result = FLEXCAN_Set_buf15_stdmask(0,0x7FE);
```

17.4.18 FLEXCAN_Tx_successful()

This function checks whether any message was transmitted.

Synopsis

```
bool FLEXCAN_Tx_successful(uint8_t 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 */
bool result = FLEXCAN_Tx_successful(0);
```

17.4.19 FLEXCAN_Tx_mailbox()

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

Synopsis

```
uint32_t FLEXCAN_Tx_mailbox(
                uint8_t  dev_num,
                uint32_t mailbox_number,
                void    *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 */
uint32_t result = FLEXCAN_Tx_mailbox(0,15,data_ptr);
```

17.4.20 FLEXCAN_Rx_mailbox()

This function gets data from the given FlexCAN mailbox.

Synopsis

```
uint32_t FLEXCAN_Rx_mailbox(
                uint8_t  dev_num,
                uint32_t mailbox_number,
                 void    *data)
```

Parameters

dev_num [in] — FlexCAN device number

mailbox_number [in] — FlexCAN message buffer index

data [out] — Pointer to output data buffer

Description

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

Return Value

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

Example

```
/* receive data from message buffer 15 */
uint32_t result = FLEXCAN_Rx_mailbox(0,15,data_ptr);
```

17.4.21 FLEXCAN_Disable_mailbox()

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

Synopsis

```
uint32_t FLEXCAN_Disable_mailbox(
                uint8_t  dev_num,
                uint32_t mailbox_number)
```

Parameters

dev_num [in] — FlexCAN device number

mailbox_number [in] — FlexCAN message buffer index

Description

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

Return Value

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

Example

```
/* disable message buffer 15 */
uint32_t result = FLEXCAN_Disable_mailbox(0,15);
```

17.4.22 FLEXCAN_Request_message()

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

Synopsis

```
uint32_t FLEXCAN_Request_message(
        uint8_t  dev_num,
        uint32_t mailbox_number,
        uint32_t 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 */
uint32_t result = FLEXCAN_Request_message(0,15,FLEXCAN_STANDARD);
```

17.4.23 FLEXCAN_Rx_message()

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

Synopsis

```
uint32_t FLEXCAN_Rx_message(
                uint8_t   dev_num,
                uint32_t  mailbox_number,
                uint32_t *identifier,
                uint32_t  format,
                uint32_t *data_len_code,
                void     *data,
                uint32_t  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 */
uint32_t result = FLEXCAN_Rx_message(0,15,&id,FLEXCAN_STANDARD,&len,
data_ptr,FLEXCAN_DISABLE);
```

17.4.24 FLEXCAN_Tx_message()

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

Synopsis

```
uint32_t FLEXCAN_Tx_message(
        uint8_t  dev_num,
        uint32_t mailbox_number,
        uint32_t identifier,
        uint32_t format,
        uint32_t data_len_code,
        void    *data)
```

Parameters

dev_num [in] — FlexCAN device number

mailbox_number [in] — FlexCAN message buffer index

identifier [in] — Message buffer ID to use

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

data_len_code [in] — Data length

data [in] — Transmitted data buffer

Description

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

Return Value

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

Example

```
/* transmit message once using mailbox 15 */
uint32_t result = FLEXCAN_Tx_message(0,15,id,FLEXCAN_STANDARD,8,data_ptr);
```

17.4.25 FLEXCAN_Read()

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

Synopsis

```
uint32_t FLEXCAN_Read(
        uint8_t    dev_num,
        uint32_t   offset,
        uint32_t  *data_ptr)
```

Parameters

dev_num [in] — FlexCAN device number

offset [in] — FlexCAN register offset

data_ptr [out] — Where to store the result

Description

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

Return Value

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

Example

```
/* Read ID of the first message buffer register */
uint32_t result = FLEXCAN_Read(0,FLEXCAN_MSG_BUFADDR_OFFSET+4,data_ptr);
```

17.4.26 FLEXCAN_Write()

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

Synopsis

```
uint32_t FLEXCAN_Write(
        uint8_t  dev_num,
        uint32_t offset,
        uint32_t value)
```

Parameters

dev_num [in] — FlexCAN device number

offset [in] — FlexCAN register offset

value [in] — 32 bit value to be written

Description

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

Return Value

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

Example

```
/* Write ID of the first message buffer register */
uint32_t result = FLEXCAN_Write(0,FLEXCAN_MSG_BUFADDR_OFFSET+4,0);
```

17.4.27 FLEXCAN_Get_status()

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

Synopsis

```
uint32_t FLEXCAN_Get_status(
        uint8_t   dev_num,
        uint32_t *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 */
uint32_t result = FLEXCAN_Get_status(0,data_ptr);
```

17.4.28 FLEXCAN_Update_message()

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

Synopsis

```
uint32_t FLEXCAN_Update_message(
                uint8_t  dev_num,
                void    *data_ptr,
                uint32_t data_len_code,
                uint32_t format,
                uint32_t mailbox_number)
```

Parameters

dev_num [in] — FlexCAN device number

data_ptr [in] — Response data

data_len_code [in] — Response data length

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

mailbox_number[in] — FlexCAN message buffer index

Description

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

Return Value

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

Example

```
/* update data in mailbox 15 used as remote response */
uint32_t result = FLEXCAN_Update_message(0,data_ptr,8,FLEXCAN_STANDARD,15);
```

17.4.29 FLEXCAN_Int_enable()

This function initializes and enables the interrupt for the specified FlexCAN message buffer.

Synopsis

```
uint32_t flexcan_int_enable(
uint8_t dev_num,
uint32_t 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_MAILBOX (wrong message buffer number)
- FLEXCAN_INT_ENABLE_FAILED (wrong interrupt vector)

Example

```
/* Enable interrupt for message buffer 5 */
uint32_t result = flexcan_int_enable(0,5);
```

17.4.30 FLEXCAN_Error_int_enable()

This function enables error, wake up, and Bus off interrupts.

Synopsis

```
uint32_t flexcan_error_int_enable(
uint8_t dev_num)
```

Parameters

dev_num [in] — FlexCAN device number

Description

The function enables error, wake up, and Bus off interrupts.

Return Value

- FLEXCAN_OK (success)
- kFlexCan_INT_ENABLE_FAILED (wrong interrupt vector)
- FLEXCAN_INT_ENABLE_FAILED (wrong interrupt vector)

Example

```
/* Enable error, wake up, and bus off interrupts */
uint32_t result = flexcan_error_int_enable(0);
```

17.4.31 FLEXCAN_Int_disable()

This function disables the interrupt for the specified FlexCAN message buffer.

Synopsis

```
uint32_t flexcan_int_disable(
uint8_t dev_num,
uint32_t 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 and disables the specified message buffer interrupt source.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)
- FLEXCAN_INT_DISABLE_FAILED (wrong interrupt vector)

Example

```
/* Disable interrupt for message buffer 5 */
uint32_t result = flexcan_int_disable(0,5);
```

17.4.32 FLEXCAN_Error_int_disable()

This function disables error, wake up, and Bus off interrupts.

Synopsis

```
uint32_t flexcan_error_int_disable(
uint8_t dev_num)
```

Parameters

dev_num [in] — FlexCAN device number

Description

The function disables error, wake up, and Bus off interrupts.

Return Value

- FLEXCAN_OK (success)
- kFlexCan_INT_DISABLE_FAILED (wrong interrupt vector)
- FLEXCAN_INT_DISABLE_FAILED (wrong interrupt vector)

Example

```
/* Disable error, wake up, and bus off interrupts */
uint32_t result = flexcan_error_int_disable(0);
```

17.4.33 FLEXCAN_Install_isr()

This function installs the interrupt service routine for the specified FlexCAN message buffer.

Synopsis

```
uint32_t flexcan_install_isr(
uint8_t dev_num,
uint32_t mailbox_number,
INT_ISR_FPTR 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 for FlexCAN message buffer Tx or Rx requests.

Note

On some systems all message buffers share the same interrupt vector. Therefore, this function installs one routine for all message buffers at once.

Return Value

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

Example

```
void flexcan_irq_handler(void * can_ptr);
/* Install interrupt service routine for message buffer 15 */
uint32_t result = flexcan_install_isr(0, 15, flexcan_irq_handler);
```

17.4.34 FLEXCAN_Install_isr_err_int()

This function installs the FlexCAN error interrupt service routine.

Synopsis

```
uint32_t flexcan_install_isr_err_int(
uint8_t dev_num,
INT_ISR_FPTR isr)
```

Parameters

dev_num [in] — FlexCAN device number

isr — Interrupt service routine address

Description

The function installs the FlexCAN error interrupt service routine.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void flexcan_irq_handler(void * can_ptr);
/* Install interrupt service routine for a FlexCAN error */
uint32_t result = flexcan_install_isr_err_int(0, flexcan_irq_handler);
```

17.4.35 FLEXCAN_Install_isr_boff_int()

This function installs the interrupt service routine for a FlexCAN bus off.

Synopsis

```
uint32_t flexcan_install_isr_boff_int(
uint8_t dev_num,
INT_ISR_FPTR isr)
```

Parameters

dev_num [in] — FlexCAN device number.

isr — Interrupt service routine address.

Description

The function installs the interrupt service routine for a FlexCAN bus off.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void flexcan_irq_handler(void * can_ptr);
/* Install interrupt service routine for a FlexCAN bus off */
uint32_t result = flexcan_install_isr_boff_int(0, flexcan_irq_handler);
```

17.4.36 FLEXCAN_Install_isr_wake_int()

This function installs the interrupt service routine for a FlexCAN wake-up.

Synopsis

```
uint32_t flexcan_install_isr_wake_int(
uint8_t dev_num,
INT_ISR_FPTR isr)
```

Parameters

dev_num [in] — FlexCAN device number

isr — Interrupt service routine address

Description

The function installs the interrupt service routine for a FlexCAN wake-up.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void flexcan_irq_handler(void * can_ptr);
/* Install interrupt service routine for a FlexCAN wake-up */
uint32_t result = flexcan_install_isr_wake_int(0, flexcan_irq_handler);
```

17.4.37 FLEXCAN_Int_status()

This function returns the FlexCAN interrupt status.

Synopsis

```
uint32_t FLEXCAN_Int_status(uint8_t 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 */
uint32_t result = FLEXCAN_Int_status(0);
```

17.5 Data Types

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

17.5.1 FLEXCAN_MSG_OBJECT_STRUCT

This structure can be used to access the FlexCAN message buffer register space directly.

```
typedef struct mcfxxxx_flexcan_msg_struct
{
  uint32_t      CONTROL;
  uint32_t      ID;
  unsigned char DATA[8];
} MCFXXXX_FCAN_MSG_STRUCT, * MCFXXXX_FCAN_MSG_STRUCT_PTR;
typedef volatile struct mcfxxxx_flexcan_msg_struct VMCFXXXX_FCAN_MSG_STRUCT;
typedef volatile struct mcfxxxx_flexcan_msg_struct * 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;
```

17.6 Error Codes

The FlexCAN driver defines the following error codes:

Table 17-1. FlexCAN driver 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
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
FLEXCAN_INVALID_ID_TYPE	Invalid ID type

17.7 Example

The FSL FlexCAN example application shows how to use FSL FlexCAN driver API functions and is provided with the MQX RTOS installation. It is located in the mqx/examples/can/flexcan directory. The source file fsl_flexcan_test.c is used for the FSL FlexCAN driver example.

Chapter 18
FSL FlexCAN Driver

18.1 Overview

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

18.2 Source Code Location

The source files for the FSL FlexCAN driver are located in source/io/can/flexcan directory. The files with prefix fsl_ are used for the FSL FlexCAN driver.

18.3 Header files

To use the FSL FlexCAN driver, include the header file fsl_flexcan_hal.h and fsl_flexcan_driver.h into your application.

18.4 API Function Reference – FSL FlexCAN Module Related Functions

This section provides function reference for the FlexCAN module driver.

NOTE

The general term "MB" refers to the Message Buffer in the FlexCAN Reference Manual terminology.

18.4.1 flexcan_set_bitrate

The function sets up all time segment values.

Synopsis

```
uint32_t flexcan_set_bitrate(
uint8_t instance,
uint32_t bitrate)
```

Parameters

- Instance--The FlexCAN instance number.
- bitrate --a FlexCAN bit rate (Bit/s) in the bit_rate_table.

Description

The function sets up all the time segment values. Those time segment values are from the table bit_rate_table and based on the bit rate in Bit/s passed in.

Return Value

- kFlexCan_OK (success)
- kFlexCan_INVALID_FREQUENCY (invalid bitrate)
- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)

Example

```
// Set FlexCAN bit rate
uint32_t result = flexcan_set_bitrate(instance, 250000);
```

18.4.2 flexcan_get_bitrate()

This function gets the FlexCAN bitrate.

Synopsis

```
uint32_t flexcan_get_bitrate(
uint8_t instance,
uint32_t *bitrate)
```

Parameters

- Instance The FlexCAN instance number.
- bitrate Pointer to a variable for returning the FlexCAN bit rate (Bit/s) in the bit_rate_table.

Description

This function is based on all the time segment values and finds out the bit rate from the table bit_rate_table.

Return Value

- kFlexCan_OK (success)
- kFlexCan_INVALID_FREQUENCY (invalid bit rate)
- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)

Example

```
// Get FlexCAN bit rate
uint32_t bitrate_get = 0;
uint32_t result = flexcan_get_bitrate(instance, &bitrate_get);
```

18.4.3 flexcan_set_mask_type()

This function sets the FlexCAN Rx masking type.

Synopsis

```
void flexcan_set_mask_type(
uint8_t instance,
flexcan_rx_mask_type_t type)
```

Parameters

- Instance The FlexCAN instance number.
- type The FlexCAN RX mask type.

Description

This function sets RX masking type as RX global mask or RX individual mask. Available mask types are:

- kFlexCanRxMask_Global (RX global mask)
- kFlexCanRxMask_Individual (RX individual mask)

Example

```
// Set FlexCAN Rx masking type
flexcan_set_mask_type(instance, kFlexCanRxMask_Global);
```

18.4.4 flexcan_set_rx_fifo_global_mask()

This function sets Rx FIFO global mask as the 11-bit standard mask or the 29-bit extended mask.

Synopsis

```
uint32_t flexcan_set_rx_fifo_global_mask(
uint8_t instance,
flexcan_mb_id_type_t id_type,
uint32_t mask)
```

Parameters

- Instance The FlexCAN instance number.
- id_type Standard ID or extended ID
- mask Mask value.

Description

This function sets Rx FIFO global mask as the 11-bit standard mask or the 29-bit extended mask. Available Rx FIFO ID types are:

- kFlexCanMbId_Std (standard ID)
- kFlexCanMbId_Ext (extended ID)

Return Value

- kFlexCan_OK (success)
- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_INVALID_ID_TYPE(invalid ID type)

Example

```
// Set Rx FIFO global mask
uint32_t result = flexcan_set_rx_fifo_global_mask(instance, kFlexCanMbId_Std, 0x7FF);
```

18.4.5 flexcan_set_rx_mb_global_mask()

This function sets Rx MB global mask as the 11-bit standard mask or the 29-bit extended mask.

Synopsis

```
uint32_t flexcan_set_rx_mb_global_mask(
```

```
uint8_t instance,
flexcan_mb_id_type_t id_type,
uint32_t mask)
```

Parameters

- Instance The FlexCAN instance number.
- id_type Standard ID or extended ID.
- mask Mask value.

Description

This function sets Rx Message Buffer global mask as the 11-bit standard mask or the 29-bit extended mask. Available Rx MB ID types are:

- kFlexCanMbId_Std (standard ID)
- kFlexCanMbId_Ext (extended ID)

Return Value

- kFlexCan_OK (success)
- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_INVALID_ID_TYPE(invalid ID type)

Example

```
// Set Rx MB global mask
uint32_t result = flexcan_set_rx_mb_global_mask(instance, kFlexCanMbId_Std, 0x123);
```

18.4.6 flexcan_set_rx_individual_mask()

This function sets Rx individual mask as the 11-bit standard mask or the 29-bit extended mask.

Synopsis

```
uint32_t flexcan_set_rx_individual_mask(
uint8_t instance,
flexcan_mb_id_type_t id_type,
uint32_t mb_idx,
uint32_t mask)
```

Parameters

- Instance The FlexCAN instance number.
- id_type Standard ID or extended ID.
- mb_idx Index of the message buffer
- mask Mask value.

Description

This function sets Rx individual ID mask as the 11-bit standard mask or the 29-bit extended mask. Available Rx individual ID mask types are:

- kFlexCanMbId_Std (standard ID)
- kFlexCanMbId_Ext (extended ID)

Return Value

- kFlexCan_OK (success)
- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_INVALID_ID_TYPE(invalid ID type)

Example

```
// Set Rx individual ID mask
uint32_t result = flexcan_set_rx_individual_mask(instance, kFlexCanMbId_Std, 0x8, 0x123);
```

18.4.7 flexcan_init()

This function initializes FlexCAN driver.

Synopsis

```
uint32_t flexcan_init(
uint8_t instance,
flexcan_config_t *data,
bool enable_err_interrupts)
```

Parameters

- Instance The FlexCAN instance number.
- data The FlexCAN platform data.
- enable_err_interrupts 1 if enable it, 0 if not.

Description

This function selects a source clock, resets FlexCAN module, sets maximum number of message buffers, initializes all message buffers as inactive, enables RX FIFO if needed, masks all mask bits, disables all MB interrupts, enables FlexCAN normal mode, and enables all the error interrupts if needed.

Return Value

- kFlexCan_OK (success)

- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_UNDEF_ERROR (data is undefined)

Example

```
// Initialize FlexCAN driver
uint32_t result = flexcan_init(instance, &flexcan1_data, FALSE);
```

18.4.8 flexcan_tx_mb_config()

This function configures FlexCAN Tx Message Buffer fields.

Synopsis

```
uint32_t flexcan_tx_mb_config(
uint8_t instance,
flexcan_config_t *data,
uint32_t mb_idx,
flexcan_mb_code_status_tx_t *cs,
uint32_t msg_id)
```

Parameters

- instance The FlexCAN instance number.
- data The FlexCAN platform data.
- mb_idx Index of the message buffer.
- cs CODE/status values (TX).
- msg_id ID of the message to transmit.

Description

This function first checks if RX FIFO is enabled. If RX FIFO is enabled, the function makes sure that the MB requested is not occupied by RX FIFO and ID filter table. Then this function sets up the Message Buffer fields, configures the Message Buffer CODE field for Tx message buffer as INACTIVE, and enables the Message Buffer interrupt. Available CODE/status values for Tx are:

- kFlexCanTX_Inactive //MB is not active
- kFlexCanTX_Abort //MB is aborted
- kFlexCanTX_Data //MB is a TX Data Frame(MB RTR must be 0)
- kFlexCanTX_Remote //MB is a TX Remote Request Frame (MB RTR must be 1)
- kFlexCanTX_Tanswer //MB is a TX Response Request Frame from //an incoming Remote Request Frame
- kFlexCanTX_NotUsed //Not used

Return Value

- kFlexCan_OK (success)
- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_INVALID_MAILBOX (invalid Message Buffer)

Example

```
// configure a Tx MB
uint32_t TX_identifier;
uint32_t TX_mailbox_num;
flexcan_config_t flexcan1_data;
uint8_t instance;

instance = 1;
flexcan1_data.num_mb = 16;
flexcan1_data.max_num_mb = 16;
flexcan1_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan1_data.is_rx_fifo_needed = FALSE;
flexcan1_data.is_rx_mb_needed = TRUE;
TX_mailbox_num = 9;
TX_identifier = 0x321;

flexcan_mb_code_status_tx_t tx_cs1;
tx_cs1.code = kFlexCanTX_Data;
tx_cs1.msg_id_type = kFlexCanMbId_Std;
tx_cs1.data_length = 1;
tx_cs1.substitute_remote = 0;
tx_cs1.remote_transmission = 0;
tx_cs1.local_priority_enable = 0;
tx_cs1.local_priority_val = 0;
uint32_t result = flexcan_tx_mb_config(instance, &flexcan1_data, TX_mailbox_num, &tx_cs1,
TX_identifier);
```

18.4.9 flexcan_send

This function sets up FlexCAN Message buffer fields to transmit data.

Synopsis

```
uint32_t flexcan_send(
uint8_t instance,
flexcan_config_t *data,
uint32_t mb_idx,
flexcan_mb_code_status_tx_t *cs,
uint32_t msg_id,
uint32_t num_bytes,
uint8_t *mb_data)
```

Parameters

- instance The FlexCAN instance number.
- data The FlexCAN platform data.
- mb_idx Index of the message buffer.
- cs CODE/status values (Tx).
- msg_id ID of the message to transmit.

- num_bytes The number of bytes in mb_data.
- mb_data Bytes of the FlexCAN message.

Description

This function first sets the MB CODE field as DATA for Tx message buffer. Then, it copies user's buffer data into the message buffer data field and waits for the Message Buffer interrupt.

Return Value

- kFlexCan_OK (success)
- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_INVALID_MAILBOX (invalid Message Buffer)
- kFlexCan_UNDEF_ERROR (Not defined)
- kFlexCan_NO_MESSAGE (No message data)

Example

```
// Configures FlexCAN Message buffer for transmitting data
uint32_t TX_identifier;
uint32_t TX_mailbox_num;
flexcan_config_t flexcan1_data;
uint8_t instance;
uint8_t  data = 23;

instance = 1;
flexcan1_data.num_mb = 16;
flexcan1_data.max_num_mb = 16;
flexcan1_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan1_data.is_rx_fifo_needed = FALSE;
flexcan1_data.is_rx_mb_needed = TRUE;
TX_mailbox_num = 9;
TX_identifier = 0x321;

flexcan_mb_code_status_tx_t tx_cs1;
tx_cs1.code = kFlexCanTX_Data;
tx_cs1.msg_id_type = kFlexCanMbId_Std;
tx_cs1.data_length = 1;
tx_cs1.substitute_remote = 0;
tx_cs1.remote_transmission = 0;
tx_cs1.local_priority_enable = 0;
tx_cs1.local_priority_val = 0;
uint32_t result = flexcan_send(instance, &flexcan1_data, TX_mailbox_num, &tx_cs1,
TX_identifier, 1, &data);
```

18.4.10 flexcan_rx_mb_config()

This function configures a FlexCAN Rx message buffer fields for receiving data.

Synopsis

```
    uint32_t flexcan_rx_mb_config(
```

```
    uint8_t instance,
    flexcan_config_t *data,
    uint32_t mb_idx,
    flexcan_mb_code_status_rx_t *cs,
    uint32_t msg_id)
```

Parameters

- instance The FlexCAN instance number.
- data The FlexCAN platform data.
- mb_idx Index of the message buffer.
- cs CODE/status values (RX).
- msg_id ID of the message to transmit.

Description

This function checks if RX FIFO is enabled. If RX FIFO is enabled, the function makes sure that the MB requested is not occupied by RX FIFO and ID filter table. Then, it sets up the message buffer fields, configures the message buffer CODE for Rx message buffer as NOT_USED, enables the Message Buffer interrupt, configures the message buffer CODE for Rx message buffer as INACTIVE, copies user's buffer data into the message buffer data field, and configures the message buffer CODE for Rx message buffer as EMPTY. Available CODE/status values for Rx are:

- kFlexCanRX_Inactive //MB is not active
- kFlexCanRX_Full // MB is full
- kFlexCanRX_Empty //MB is active and empty
- kFlexCanRX_Overrun // MB is being overwritten into a full buffer
- kFlexCanRX_Busy // FlexCAN is updating the contents of the MB // The CPU must not access the MB
- kFlexCanRX_Ranswer // A frame was configured to recognize a Remote Request Frame / and transmit a Response Frame in return
- kFlexCanRX_NotUsed // Not used

Return Value

- kFlexCan_OK (success)
- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_INVALID_MAILBOX (invalid Message Buffer)
- kFlexCan_UNDEF_ERROR (Not defined)
- kFlexCan_NO_MESSAGE (No message data)

Example

```
// Configure RX MB fields
uint32_t RX_identifier;
uint32_t RX_mailbox_num;
flexcan_config_t flexcan1_data;
uint8_t instance;
```

```
instance = 1;
flexcan1_data.num_mb = 16;
flexcan1_data.max_num_mb = 16;
flexcan1_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan1_data.is_rx_fifo_needed = FALSE;
flexcan1_data.is_rx_mb_needed = TRUE;
RX_mailbox_num = 8;
RX_identifier = 0x123;

flexcan_mb_code_status_rx_t rx_cs1;
rx_cs1.code = kFlexCanRX_Ranswer;
rx_cs1.msg_id_type = kFlexCanMbId_Std;
rx_cs1.data_length = 1;
rx_cs1.substitute_remote = 0;
rx_cs1.remote_transmission = 0;
rx_cs1.local_priority_enable = 0;
rx_cs1.local_priority_val = 0;
result = flexcan_rx_mb_config(instance, &flexcan1_data, RX_mailbox_num, &rx_cs1,
RX_identifier);
```

18.4.11 flexcan_rx_fifo_config()

This function configures the FlexCAN RX FIFO fields.

Synopsis

```
uint32_t flexcan_rx_fifo_config(
uint8_t instance,
flexcan_config_t *data,
flexcan_rx_fifo_id_element_format_t id_format,
flexcan_id_table_t *id_filter_table)
```

Parameters

- instance The FlexCAN instance number.
- data The FlexCAN platform data.
- id_format The format of the Rx FIFO ID Filter Table Elements.
- id_filter_table The ID filter table elements which contain RTR bit, IDE bit, and RX message ID.

Description

This function configures Rx FIFO ID filter table elements and enables Rx FIFO interrupts. Available Rx FIFO ID element formats are:

- kFlexCanRxFifoIdElementFormat_A // One full ID (standard and extended) per ID Filter // Table element.
- kFlexCanRxFifoIdElementFormat_B // Two full standard IDs or two partial 14-bit // (standard and extended) IDs per ID Filter Table element.
- kFlexCanRxFifoIdElementFormat_C //Four partial 8-bit Standard IDs per ID Filter Table // element.
- kFlexCanRxFifoIdElementFormat_D // All frames rejected.

Return Value

- kFlexCan_OK (success)
- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_INVALID_MAILBOX (invalid Message Buffer)
- kFlexCan_UNDEF_ERROR (Not defined)
- kFlexCan_NO_MESSAGE (No message data)

Example

```
// Configure RX FIFO fields
flexcan_config_t flexcan1_data;
flexcan_id_table_t id_table;
uint8_t instance;

instance = 1;
flexcan1_data.num_mb = 16;
flexcan1_data.max_num_mb = 16;
flexcan1_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan1_data.is_rx_fifo_needed = FALSE;
flexcan1_data.is_rx_mb_needed = TRUE;

uint8_t result = flexcan_rx_fifo_config(instance, &flexcan1_data,
kFlexCanRxFifoIdElementFormat_A, &id_table);
```

18.4.12 flexcan_start_receive()

This function starts receiving data.

Synopsis

```
uint32_t flexcan_start_receive(
uint8_t instance,
flexcan_config_t *data,
uint32_t mb_idx,
uint32_t msg_id,
uint32_t receiveDataCount,
bool *is_rx_mb_data,
bool *is_rx_fifo_data,
flexcan_mb_t *rx_mb,
flexcan_mb_t *rx_fifo)
```

Parameters

- instance The FlexCAN instance number.
- data The FlexCAN platform data.
- mb_idx Index of the message buffer.
- msg_id ID of the message to transmit.
- receiveDataCount The number of data to be received.
- is_rx_mb_data Checking if the data received is from Rx MB
- is_rx_fifo_data Checking if the data received is from Rx FIFO.

- rx_mb The FlexCAN receive message buffer data.
- rx_fifo The FlexCAN receive FIFO data.

Description

This function locks Rx MB or Rx FIFO after getting an interrupt for an Rx MB or an Rx FIFO, gets the Rx MB or Rx FIFO field values, and unlocks the Rx MB or the Rx FIFO.

Return Value

- kFlexCan_OK (success)
- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)
- kFlexCan_INVALID_MAILBOX (invalid Message Buffer)
- kFlexCan_UNDEF_ERROR (Not defined)
- kFlexCan_NO_MESSAGE (No message data)

Example

```
// Start receiving data
uint32_t RX_identifier;
uint32_t RX_mailbox_num;
flexcan_config_t flexcan1_data;
uint8_t instance;
flexcan_mb_t rx_fifo;
flexcan_mb_t rx_mb;
bool is_rx_mb_data = FALSE;
bool is_rx_fifo_data = FALSE;

instance = 1;
flexcan1_data.num_mb = 16;
flexcan1_data.max_num_mb = 16;
flexcan1_data.num_id_filters = kFlexCanRxFifoIDFilters_8;
flexcan1_data.is_rx_fifo_needed = FALSE;
flexcan1_data.is_rx_mb_needed = TRUE;
RX_mailbox_num = 8;
RX_identifier = 0x123;

flexcan_mb_code_status_rx_t rx_cs1;
rx_cs1.code = kFlexCanRX_Ranswer;
rx_cs1.msg_id_type = kFlexCanMbId_Std;
rx_cs1.data_length = 1;
rx_cs1.substitute_remote = 0;
rx_cs1.remote_transmission = 0;
rx_cs1.local_priority_enable = 0;
rx_cs1.local_priority_val = 0;

uint8_t result = flexcan_start_receive(instance, &flexcan1_data, RX_mailbox_num,
RX_identifier, 1, &is_rx_mb_data, &is_rx_fifo_data, &rx_mb, &rx_fifo);
```

18.4.13 flexcan_shutdown()

The function shuts down a FlexCAN instance.

Synopsis

```
uint32_t flexcan_shutdown(uint8_t instance)
```

Parameters

- Instance The FlexCAN instance number.

Description

This function disables FlexCAN.

Return Value

- kFlexCan_OK (success)
- kFlexCan_INVALID_ADDRESS (invalid FlexCAN base address)

Example

```
/* Shutdown FlexCAN */
uint8_t instance = 1;
uint32_t result = flexcan_shutdown(instance);
```

18.4.14 flexcan_irq_handler()

The function is the interrupt handler for FlexCAN.

Synopsis

```
void flexcan_irq_handler(void * can_ptr)
```

Parameters

- can_ptr point to a FlexCAN instance.

Description

The function is the interrupt handler for FlexCAN.

Example

```
// Install ISR
uint8_t instance = 1;
uint32_t mb_idx = 8;
uint32_t result = flexcan_install_isr(instance, mb_idx, flexcan_irq_handler);
```

18.4.15 FLEXCAN_Int_enable()

This function initializes and enables the interrupt for the specified FlexCAN message buffer.

Synopsis

```
uint32_t flexcan_int_enable(
uint8_t dev_num,
uint32_t 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_MAILBOX (wrong message buffer number)
- FLEXCAN_INT_ENABLE_FAILED (wrong interrupt vector)

Example

```
/* Enable interrupt for message buffer 5 */
uint32_t result = flexcan_int_enable(0,5);
```

18.4.16 FLEXCAN_Int_disable()

This function disables the interrupt for the specified FlexCAN message buffer.

Synopsis

```
uint32_t flexcan_int_disable(
uint8_t dev_num,
uint32_t 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 and disables the specified message buffer interrupt source.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_MAILBOX (wrong message buffer number)
- FLEXCAN_INT_DISABLE_FAILED (wrong interrupt vector)

Example

```
/* Disable interrupt for message buffer 5 */
uint32_t result = flexcan_int_disable(0,5);
```

18.4.17 FLEXCAN_Install_isr()

This function installs the interrupt service routine for the specified FlexCAN message buffer.

Synopsis

```
uint32_t flexcan_install_isr(
uint8_t dev_num,
uint32_t mailbox_number,
INT_ISR_FPTR 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 for FlexCAN message buffer Tx or Rx requests.

Note

On some systems all message buffers share the same interrupt vector. Therefore, this function installs one routine for all message buffers at once.

Return Value

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

Example

```
void flexcan_irq_handler(void * can_ptr);
/* Install interrupt service routine for message buffer 15 */
uint32_t result = flexcan_install_isr(0, 15, flexcan_irq_handler);
```

18.4.18 flexcan_uninstall_isr()

This function uninstalls the interrupt service routine for the specified FlexCAN message buffer.

Synopsis

```
uint32_t flexcan_uninstall_isr(
uint8_t dev_num)
```

Parameters

- dev_num – FlexCAN device number

Description

The function uninstalls the interrupt service routine FlexCAN message buffer Tx or Rx requests.

NOTE

On some systems all message buffers share the same interrupt vector. Therefore, this function uninstalls one routine for all message buffers at once.

Return Value

- kFlexCan _OK (success)
- kFlexCan _INVALID_ADDRESS (wrong device number)
- kFlexCan _INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void flexcan_irq_handler(void * can_ptr);
```

```
/* Uninstall interrupt service routine */
uint32_t result = flexcan_uninstall_isr(0);
```

18.4.19 FLEXCAN_Error_int_enable()

This function enables error, wake up, and Bus off interrupts.

Synopsis

```
uint32_t flexcan_error_int_enable(
uint8_t dev_num)
```

Parameters

dev_num [in] — FlexCAN device number

Description

The function enables error, wake up, and Bus off interrupts.

Return Value

- FLEXCAN_OK (success)
- kFlexCan_INT_ENABLE_FAILED (wrong interrupt vector)
- FLEXCAN_INT_ENABLE_FAILED (wrong interrupt vector)

Example

```
/* Enable error, wake up, and bus off interrupts */
uint32_t result = flexcan_error_int_enable(0);
```

18.4.20 FLEXCAN_Error_int_disable()

This function disables error, wake up, and Bus off interrupts.

Synopsis

```
uint32_t flexcan_error_int_disable(
uint8_t dev_num)
```

Parameters

dev_num [in] — FlexCAN device number

Description

The function disables error, wake up, and Bus off interrupts.

Return Value

- FLEXCAN_OK (success)
- kFlexCan_INT_DISABLE_FAILED (wrong interrupt vector)
- FLEXCAN_INT_DISABLE_FAILED (wrong interrupt vector)

Example

```
/* Disable error, wake up, and bus off interrupts */
uint32_t result = flexcan_error_int_disable(0);
```

18.4.21 FLEXCAN_Install_isr_err_int()

This function installs the FlexCAN error interrupt service routine.

Synopsis

```
uint32_t flexcan_install_isr_err_int(
uint8_t dev_num,
INT_ISR_FPTR isr)
```

Parameters

dev_num [in] — FlexCAN device number

isr — Interrupt service routine address

Description

The function installs the FlexCAN error interrupt service routine.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void flexcan_irq_handler(void * can_ptr);
/* Install interrupt service routine for a FlexCAN error */
uint32_t result = flexcan_install_isr_err_int(0, flexcan_irq_handler);
```

18.4.22 flexcan_uninstall_isr_err_int()

This function uninstalls the interrupt service routine for a FlexCAN error.

Synopsis

```
uint32_t flexcan_uninstall_isr_err_int(
uint8_t dev_num)
```

Parameters

- dev_num – FlexCAN device number

Description

The function uninstalls the interrupt service routine for a FlexCAN error.

Return Value

- kFlexCan _OK (success)
- kFlexCan _INVALID_ADDRESS (wrong device number)
- kFlexCan _INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void flexcan_irq_handler(void * can_ptr);
/* Uninstall interrupt service routine for a FlexCAN error */
uint32_t result = flexcan_uninstall_isr_err_int(0);
```

18.4.23 FLEXCAN_Install_isr_boff_int()

This function installs the interrupt service routine for a FlexCAN bus off.

Synopsis

```
uint32_t flexcan_install_isr_boff_int(
uint8_t dev_num,
INT_ISR_FPTR isr)
```

Parameters

dev_num [in] — FlexCAN device number.

isr — Interrupt service routine address.

Description

The function installs the interrupt service routine for a FlexCAN bus off.

Return Value

- FLEXCAN_OK (success)

- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void flexcan_irq_handler(void * can_ptr);
/* Install interrupt service routine for a FlexCAN bus off */
uint32_t result = flexcan_install_isr_boff_int(0, flexcan_irq_handler);
```

18.4.24 flexcan_uninstall_isr_boff_int()

This function uninstalls the interrupt service routine for a FlexCAN bus off.

Synopsis

```
uint32_t flexcan_uninstall_isr_boff_int(
uint8_t dev_num)
```

Parameters

- dev_num – FlexCAN device number

Description

The function uninstalls the interrupt service routine for a FlexCAN bus off.

Return Value

- kFlexCan _OK (success)
- kFlexCan _INVALID_ADDRESS (wrong device number)
- kFlexCan _INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void flexcan_irq_handler(void * can_ptr);
/* Uninstall interrupt service routine for a FlexCAN bus off */
uint32_t result = flexcan_uninstall_isr_boff_int(0);
```

18.4.25 FLEXCAN_Install_isr_wake_int()

This function installs the interrupt service routine for a FlexCAN wake-up.

Synopsis

```
uint32_t flexcan_install_isr_wake_int(
uint8_t dev_num,
INT_ISR_FPTR isr)
```

Parameters

dev_num [in] — FlexCAN device number

isr — Interrupt service routine address

Description

The function installs the interrupt service routine for a FlexCAN wake-up.

Return Value

- FLEXCAN_OK (success)
- FLEXCAN_INVALID_ADDRESS (wrong device number)
- FLEXCAN_INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void flexcan_irq_handler(void * can_ptr);
/* Install interrupt service routine for a FlexCAN wake-up */
uint32_t result = flexcan_install_isr_wake_int(0, flexcan_irq_handler);
```

18.4.26 flexcan_uninstall_isr_wake_int()

This function uninstalls the interrupt service routine for a FlexCAN wake-up.

Synopsis

```
uint32_t flexcan_uninstall_isr_wake_int(
uint8_t dev_num)
```

Parameters

- dev_num – FlexCAN device number

Description

The function uninstalls the interrupt service routine for a FlexCAN wake-up.

Return Value

- kFlexCan _OK (success)
- kFlexCan _INVALID_ADDRESS (wrong device number)
- kFlexCan _INT_INSTALL_FAILED (wrong interrupt vector)

Example

```
void flexcan_irq_handler(void * can_ptr);
/* Uninstall interrupt service routine for a FlexCAN wake-up */
uint32_t result = flexcan_uninstall_isr_wake_int(0);
```

18.5 Data Types

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

18.5.1 flexcan_mb_code_status_tx

This structure configures the FlexCAN Tx message buffer CODE and status field.

```
    typedef struct flexcan_mb_code_status_tx {
    flexcan_mb_code_tx_t code;          // MB code for TX buffers
    flexcan_mb_id_type_t msg_id_type;   // Type of message ID (standard or
                                        //  extended)
    uint32_t data_length;               // Length of Data in Bytes
    uint32_t substitute_remote;         // Substitute remote request (used
                                        //  only in extended format)
    uint32_t remote_transmission;       // Remote transmission request
    bool local_priority_enable;         // 1 if enable it; 0 if disable it
    uint32_t local_priority_val;        // Local priority value [0..2]
} flexcan_mb_code_status_tx_t;


flexcan_mb_code_status_tx_t tx_cs1;
```

18.5.2 flexcan_mb_code_status_rx

This structure configures the FlexCAN Rx message buffer code and status field.

```
 typedef struct flexcan_mb_code_status_rx {
    flexcan_mb_code_rx_t code;          // MB code for RX buffers
    flexcan_mb_id_type_t msg_id_type;   // Type of message ID (standard or
                                        //  extended)
    uint32_t data_length;               // Length of Data in Bytes
    uint32_t substitute_remote;         // Substitute remote request (used
                                        //!  only in extended format)
    uint32_t remote_transmission;       // Remote transmission request
    bool local_priority_enable;         // 1 if enable it; 0 if disable it
    uint32_t local_priority_val;        // Local priority value [0..2]
} flexcan_mb_code_status_rx_t;


flexcan_mb_code_status_rx_t rx_cs1;
```

18.5.3 flexcan_mb

This structure configures the FlexCAN Rx MB or Rx FIFO fields.

```
typedef struct flexcan_mb {
    uint32_t cs;                        // Code and Status
    uint32_t msg_id;                    // Message Buffer ID
    uint8_t data[kFlexCanMessageSize];  // bytes of the FlexCAN message
} flexcan_mb_t;



flexcan_mb_t rx_fifo;
flexcan_mb_t rx_mb;
```

18.5.4 flexcan_config

This structure configures FlexCAN.

```
typedef struct flexcan_config {
    uint32_t num_mb;             // The number of Message Buffers needed
    uint32_t max_num_mb;         // The maximum number of Message Buffers
    flexcan_rx_fifo_id_filter_num_t num_id_filters;
                                 // The number of RX FIFO ID filters needed
    bool is_rx_fifo_needed;      // 1 if need it; 0 if not
    bool is_rx_mb_needed;        // 1 if need it; 0 if not
} flexcan_config_t;


flexcan_config_t flexcan1_data;
```

18.5.5 flexcan_time_segment

This structure sets up time segments.

```
typedef struct flexcan_time_segment {
    uint32_t propseg;     // Propagation segment
    uint32_t pseg1;       // Phase segment 1
    uint32_t pseg2;       // Phase segment 2
    uint32_t pre_divider; // Clock pre divider
    uint32_t rjw;         // Resync jump width
} flexcan_time_segment_t;


flexcan_time_segment_t time_seg;
```

18.6 Error Codes

The FlexCAN driver defines the following error codes:

Table 18-1. FlexCAN driver 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
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
FLEXCAN_INVALID_ID_TYPE	Invalid ID type

18.7 Example

The FSL FlexCAN example application shows how to use FSL FlexCAN driver API functions and is provided with the MQX RTOS installation. It is located in the mqx/examples/can/flexcan directory. The source file fsl_flexcan_test.c is used for the FSL FlexCAN driver example.

Chapter 19
NAND Flash Driver

19.1 Overview

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

19.2 Source Code Location

Table 19-1. 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	`sourceio/nadflash/nand_devices`

19.3 Header Files

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

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

19.4 Hardware Supported

The MQX NAND Flash driver currently supports Freescale microprocessors containing NAND Flash Controller (NFC) peripheral module only. However, the driver can be modified to access NAND Flash memory devices directly which is a software driven solution.

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

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

User has to describe the structure of the NAND Flash memory to be supported. See NANDFLASH_INFO_STRUCT. It also has to pass this structure as an initialization parameter during the driver installation. See NANDFLASH_INIT_STRUCT for a detailed description.

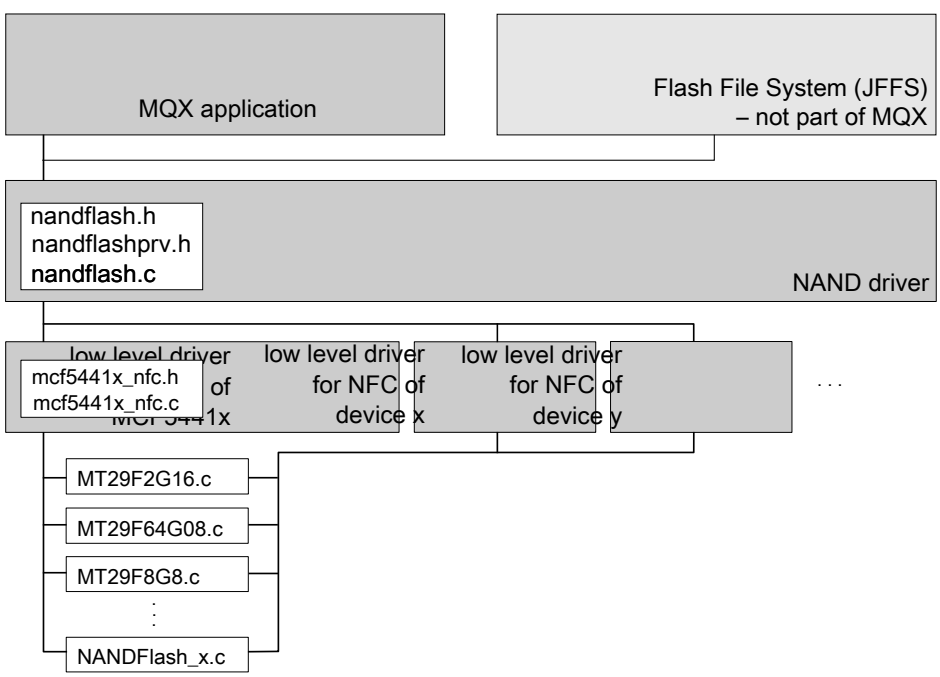

Figure 19-1. MQX NAND Flash Driver Layers

19.5 Driver Services

NAND Flash driver provides the following I/O services.

Table 19-2. NAND Flash driver 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()

19.6 Installing NAND Flash Driver

NAND Flash driver provides the **_io_nandflash_install()** installation function that either the BSP or the application calls. The function fills in the configuration structures and calls **_io_dev_install_ext()** internally.

In the BSPs distributed with Freescale MQX installation, the **_io_nandflash_install()** installation function is called from *init_bsp.c*. The functionality can be enabled or disabled by setting **BSPCFG_ENABLE_NANDFLASH** configuration option to 1 or 0 in *user_config.h*.

Example

```
result = _io_nandflash_install(&_bsp_nandflash_init);
```

The `_bsp_nandflash_init` is an initialization structure of the NANDFLASH_INIT_STRUCT type containing initialization data for the NAND Flash driver.

19.6.1 NANDFLASH_INIT_STRUCT

This structure contains initialization data and is passed to the NAND Flash driver installation function.

Synopsis

```
struct nandflash_init_struct {
   char                        *ID_PTR;
   uint32_t (_CODE_PTR_         INIT)(struct io_nandflash_struct *);
   void (_CODE_PTR_             DEINIT)(struct io_nandflash_struct *);
   uint32_t (_CODE_PTR_         CHIP_ERASE)(struct io_nandflash_struct *);
   uint32_t (_CODE_PTR_         BLOCK_ERASE)(struct io_nandflash_struct *,
     uint32_t, bool);
   uint32_t (_CODE_PTR_         PAGE_READ)(struct io_nandflash_struct *,
     unsigned char*, uint32_t, uint32_t);
   uint32_t (_CODE_PTR_         PAGE_PROGRAM)(struct io_nandflash_struct *,
     unsigned char*, uint32_t, uint32_t);
   uint32_t (_CODE_PTR_         WRITE_PROTECT)(struct io_nandflash_struct *,
     bool);
   uint32_t (_CODE_PTR_         IS_BLOCK_BAD)(struct io_nandflash_struct *,
     uint32_t);
   uint32_t (_CODE_PTR_         MARK_BLOCK_AS_BAD)(struct io_nandflash_struct *,
     uint32_t);
   _mqx_int (_CODE_PTR_         IOCTL)(IO_NANDFLASH_STRUCT_PTR, _mqx_uint, void*);
   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;
   uint32_t                     ECC_SIZE;
   _mqx_uint                    WRITE_VERIFY;
   void                        *DEVICE_SPECIFIC_DATA;
} NANDFLASH_INIT_STRUCT, * NANDFLASH_INIT_STRUCT_PTR;
```

Parameters

- ID_PTR — Pointer to a string which identifies the device for **fopen()**.
- INIT— Pointer to the function which initializes the NAND flash device (low-level function).
- DEINIT— Pointer to the function which disables the NAND flash device (low-level function).
- CHIP_ERASE — Pointer to the function which erases the entire NAND flash (low-level function).
- SECTOR_ERASE— Pointer to the function which erases a flash sector (low-level function).
- BLOCK_ERASE— Pointer to the function which erases one NAND flash block (low-level function).
- PAGE_READ — Pointer to the function which reads pages of the NAND flash (low-level function).
- PAGE_PROGRAM — Pointer to the function which programs pages of the NAND flash (low-level function).
- WRITE_PROTECT — Pointer to the function which disables/enables writing to the NAND flash (low-level function).
- IS_BLOCK_BAD — Pointer to the function that checks if the defined block is bad (low-level function).

- MARK_BLOCK_AS_BAD — Pointer to the function which marks the defined block as bad (low-level function).
- IOCTL — Optional function for device specific commands.
- NANDFLASH_INFO_PTR — Pointer to the structure which provides an organization of the NAND flash device. See NANDFLASH_INFO_STRUCT.
- VIRTUAL_PAGE_SIZE — The size of one virtual page in bytes. One Physical page can be divided into several virtual pages if supported by the NAND Flash Controller. Virtual page is the smallest unit a block device can work with. This value is typically defined in the BSP (BSP_VIRTUAL_PAGE_SIZE).
- NUM_VIRTUAL_PAGES — The number of NAND Flash virtual pages. This value is set by the _io_nandflash_install function.
- PHY_PAGE_SIZE_TO_VIRTUAL_PAGE_SIZE_RATIO — The ratio between the physical page size and the virtual page size. This value is set by the _io_nandflash_install function.
- ECC_SIZE — The number of ECC correction bits per one virtual page. This value is typically defined in the BSP (BSP_ECC_SIZE).
- WRITE_VERIFY — When finished programming, a comparison of data should be made to verify that the write has worked correctly.
- DEVICE_SPECIFIC_DATA — The address of device-specific structure.

Example of nandflash init structure for NFC of MCF5441x device and MT29F2G16 NAND Flash memory:

```
const NANDFLASH_INIT_STRUCT _bsp_nandflash_init =
{
   /* NAME                 */   "nandflash:",
   /* INIT                 */   nfc_init,
   /* DEINIT               */   nfc_deinit,
   /* CHIP_ERASE           */   nfc_erase_flash,
   /* BLOCK_ERASE          */   nfc_erase_block,
   /* PAGE_READ            */   nfc_read_page,
   /* PAGE_PROGRAM         */   nfc_write_page,
   /* WRITE_PROTECT        */   NULL,
   /* IS_BLOCK_BAD         */   nfc_check_block,
   /* MARK_BLOCK_AS_BAD    */   nfc_mark_block_as_bad,
   /* IOCTL                */   nfc_ioctl,
   /* NANDFLASH_INFO_PTR   */   _MT29F2G16_organization_16bit,
   /* VIRTUAL_PAGE_SIZE    */   512,
   /* NUM_VIRTUAL_PAGES    */   0,
   /* PHY_PAGE_SIZE_TO_VIRTUAL_PAGE_SIZE_RATIO */
                                4,
   /* ECC_SIZE             */   4, /* 4-error correction bits (8 ECC bytes) */
   /* WRITE_VERIFY         */   0,
   /* DEVICE_SPECIFIC_DATA */   0
   };
```

All *nfc_xxx* functions are NFC module-dependent low level routines defined in *source/io/nandflash/nfc* subdirectory.

19.6.2 NANDFLASH_INFO_STRUCT

This structure contains information about a particular NAND Flash memory device.

Synopsis

```
struct nandflash_info_struct {
   _mem_size        PHY_PAGE_SIZE;
   _mem_size        SPARE_AREA_SIZE;
   _mem_size        BLOCK_SIZE;
   _mqx_uint        NUM_BLOCKS;
   _mqx_uint        WIDTH;
} NANDFLASH_INFO_STRUCT, * 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
};
```

19.7 NFC Peripheral Module-Specific Low Level Routines

NAND Flash driver refers to low-level functions which implement NAND flash atomic operations. These functions are part of the MQX release for all supported NFCs. The user passes pointers to these low-level functions in the `NANDFLASH_INIT_STRUCT` when installing the NAND Flash driver.

The functions are located in NFC-specific subdirectory in *source/io/nandflash/nfc*.

19.7.1 Init Function

This function initializes the NAND flash device.

Synopsis

```
uint32_t (_CODE_PTR_  INIT)(
           IO_NANDFLASH_STRUCT_PTR nandflash_ptr)
```

Parameters

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

19.7.2 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.

19.7.3 Chip Erase Function

This function erases the entire NAND flash device.

Synopsis

```
uint32_t (_CODE_PTR_ CHIP_ERASE)(
           IO_NANDFLASH_STRUCT_PTR nandflash_ptr)
```

Parameters

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

19.7.4 Block Erase Function

This function erases one NAND flash block.

Synopsis

```
uint32_t (_CODE_PTR_ BLOCK_ERASE)(
           IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
        uint32_t                   block_number,
        bool                       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.

19.7.5 Page Read Function

This function reads pages of the NAND flash.

Synopsis

```
uint32_t (_CODE_PTR_ PAGE_READ)(
           IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
        unsigned char             *to_ptr,
        uint32_t                   page_number,
        uint32_t                   page_count)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.
- *to_ptr [OUT]* — Where to copy the data to.
- *page_number [IN]* — Page number where to start reading.
- *page_count [IN]* — The amount of pages to be read.

19.7.6 Page Program Function

This function programs the pages of the NAND flash.

Synopsis

```
uint32_t (_CODE_PTR_ PAGE_PROGRAM)(
           IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
        unsigned char             *from_ptr,
        uint32_t                   page_number,
        uint32_t                   page_count)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.
- *from_ptr [IN]* — Where to copy the data from.
- *page_number [IN]* — Page number where to start writing.
- *page_count [IN]* — The number of pages to be programed.

19.7.7 Write Protect Function

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

Synopsis

```
uint32_t (_CODE_PTR_ WRITE_PROTECT)(
           IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
           bool                    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.

19.7.8 Is Block Bad Function

This function checks if the defined block is bad.

Synopsis

```
uint32_t (_CODE_PTR_ IS_BLOCK_BAD)(
           IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
           uint32_t                block_number)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.
- *block_number [IN]* — The block number to be checked.

19.7.9 Mark Block as Bad Function

This function is called to mark the defined block as bad.

Synopsis

```
uint32_t (_CODE_PTR_ MARK_BLOCK_AS_BAD)(
           IO_NANDFLASH_STRUCT_PTR nandflash_ptr,
           uint32_t                block_number)
```

Parameters

- *nandflash_ptr [IN]* — The device handle.
- *block_number [IN]* — The block number to be marked as bad.

19.8 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*.

Table 19-3. I/O control commands

Command	Description	Parameters
NANDFLASH_IOCTL_GET_PHY_PAGE_SIZE	Gets the NAND Flash physical page size.	*param_ptr* - pointer to uint32_t
NANDFLASH_IOCTL_GET_SPARE_AREA_SIZE	Gets the NAND Flash spare area size.	*param_ptr* - pointer to uint32_t
NANDFLASH_IOCTL_GET_BLOCK_SIZE	Gets the NAND Flash block size.	*param_ptr* - pointer to uint32_t
NANDFLASH_IOCTL_GET_NUM_BLOCKS	Gets the total number of NAND Flash blocks.	*param_ptr* - pointer to uint32_t
NANDFLASH_IOCTL_GET_WIDTH	Gets the NAND Flash width.	*param_ptr* - pointer to uint32_t

Table continues on the next page...

Table 19-3. I/O control commands (continued)

Command	Description	Parameters
NANDFLASH_IOCTL_GET_NUM_VIRT_PAGES	Gets the total number of virtual pages.	*param_ptr* - pointer to uint32_t
NANDFLASH_IOCTL_GET_VIRT_PAGE_SIZE	Gets the size of one virtual page.	*param_ptr* - pointer to uint32_t
NANDFLASH_IOCTL_ERASE_BLOCK	Erases the specified block of the NAND Flash.	*param_ptr* - pointer to uint32_t
NANDFLASH_IOCTL_ERASE_CHIP	Erases the whole NAND Flash.	none (NULL)
NANDFLASH_IOCTL_WRITE_PROTECT	Write-enable or write-protect the NAND Flash device.	*param_ptr* - pointer to bool
NANDFLASH_IOCTL_CHECK_BLOCK	Checks if the defined NAND Flash block is bad or not. The block number (uint32_t) is passed as a parameter. Returned values can be: NANDFLASHERR_BLOCK_NOT_BAD, NANDFLASHERR_BLOCK_BAD, or NANDFLASHERR_TIMEOUT	*param_ptr*- pointer to uint32_t
NANDFLASH_IOCTL_MARK_BLOCK_AS_BAD	Marks the defined NAND Flash block as bad.	*param_ptr* - pointer to uint32_t
NANDFLASH_IOCTL_GET_BAD_BLOCK_TABLE	Checks all NAND Flash blocks and gets the bad block table (field of 8-bit values, length equals to the number of NAND Flash blocks, 0 = bad block, 1 = not a bad block).	pointer to uint8_t field of size equal to the number of blocks
NANDFLASH_IOCTL_GET_ID	Gets NAND Flash ID.	*param_ptr* - pointer to uint32_t
NANDFLASH_IOCTL_ERASE_BLOCK_FORCE	Forces block erase in case the block is marked as bad.	*param_ptr* - pointer to uint32_t

19.9 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.

19.10 Error Codes

This section describes all error codes that can be returned by the NAND Flash driver. Error codes are defined in *nandflash.h*.

Table 19-4. Error codes

Error Code	Description
NANDFLASHERR_NO_ERROR	Operation successful.
NANDFLASHERR_ECC_FAILED	Returned when the ECC engine finds that the read page cannot be corrected.
NANDFLASHERR_ECC_CORRECTED	Returned when the ECC engine corrected errors is the read page.
NANDFLASHERR_ERASE_FAILED	Returned when the erasing process fails.
NANDFLASHERR_WRITE_FAILED	Returned when writing to the NAND Flash fails.
NANDFLASHERR_TIMEOUT	Returned when any operation with the NAND Flash is timed-out.
NANDFLASHERR_BLOCK_BAD	Returned when the specified block is bad.
NANDFLASHERR_BLOCK_NOT_BAD	Returned when the specified block is not bad.
NANDFLASHERR_INFO_STRUC_MISSING	Returned when the NANDFLASH_INFO_STRUCT is not available for the driver (not defined manually and simultaneously not possible to create from the NAND ID read out of the NAND Flash).
NANDFLASHERR_IMPROPER_ECC_SIZE	Returned when the sum of virtual page size (incl. ECC bytes) per one physical page is not greater than the physical page size plus the number of physical spare bytes.

Chapter 20
DAC Driver

20.1 Overview

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

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

20.2 Source Code Location

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

20.3 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.

20.4 API Function Reference

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

20.4.1 DAC_Init()

This function (re)initializes the DAC module.

Synopsis

```
LDD_TDeviceDataPtr DAC_Init (
    /* [IN] Pointer to the RTOS device structure. */
    LDD_RTOS_TDeviceDataPtr     RTOSDeviceData
);
```

Parameters

`RTOSDeviceData` *[in]* — Pointer to the private device structure. This pointer is passed to all callback events as parameter.

Description

Initializes the device according to the design time configuration properties. Allocates memory for the device data structure. This method can be called only once. Before the second call of DAC_Init(), the DAC_Deinit() must be called first.

Return Value

`LDD_TDeviceDataPtr` — Pointer to the dynamically allocated private structure or NULL if there was an error.

Example

The following example shows how to initialize the DAC module.

```
/* DAC callback function prototypes */
void DAC_BufferStartCallBack(LDD_RTOS_TDeviceDataPtr DeviceData);
void DAC_BufferWattermarkCallBack(LDD_RTOS_TDeviceDataPtr DeviceData);
void DAC_BufferEndCallBack(LDD_RTOS_TDeviceDataPtr DeviceData);
/* DAC init structure */
const LDD_RTOS_TDeviceData  DAC_RTOS_DeviceData =
{
/* DAC device number               */  DAC_1,
/* DAC reference selection         */  DAC_PDD_V_REF_EXT,
/* DAC trigger mode                */  DAC_PDD_HW_TRIGGER,
/* DAC buffer mode                 */  LDD_DAC_BUFFER_NORMAL_MODE,
/* DAC buffer start callback       */  DAC_BufferStartCallBack,
/* DAC buffer watermark callback   */  DAC_BufferWattermarkCallBack,
/* DAC buffer end callback         */  DAC_BufferEndCallBack
};
/* Initialize DAC device */
if (NULL == (DAC_DevicePtr = DAC_Init((const
LDD_RTOS_TDeviceDataPtr)&DAC_RTOS_DeviceData)))
```

```
{
    printf("DAC device initialization failed\n");
}
```

20.4.2 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

20.4.3 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");
}
```

20.4.4 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, and so on. 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);
```

20.4.5 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;
}
```

20.4.6 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.

20.4.7 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;
}
```

20.4.8 DAC_SetValue()

This function sets the DAC output value.

Synopsis

```
LDD_TError DAC_SetValue (
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr  DeviceData,
    /* [IN] User data */
    LDD_DAC_TData       Data
);
```

Parameters

`DeviceData [in]` — Device data structure pointer.

`Data [in]` — Device data structure pointer.

Description

Sets the DAC output voltage to the specified value. This method is used when data buffering is not required. The 12-bit right justified format is assumed and no data transformation (shifting or scaling) is done in the driver.

Return Value

- DAC_ERROR_OK — OK

Example

The following example shows how to set DC value on the DAC device.

```
DAC_Error = DAC_SetValue (DAC_DevicePtr, (LDD_DAC_TData)2048);
```

20.4.9 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. */
    uint16_t           *DataArrayPtr,
    /* [IN] Length of user data array which should be written to data buffer. */
    uint8_t             DataArrayLength,
    /* [IN] Index of first written data buffer register. */
    uint8_t             StartBufferReg
);
```

Parameters

`DeviceData [in]` — Device data structure pointer.

`DataArrayPtr [in]` — Pointer to array containing user data.

`DataArrayLength [in]` — Length of user data array which should be written to data buffer.

`StartBufferReg [in]` — Index of first written data buffer register.

Description

Writes an array of data words to the data buffer registers. Array is defined by a pointer to the start address and by length. First written data buffer register is defined by an index. The rest of the array is written to registers with increasing index. If the length of array exceeds number of registers between the first written register and the last register at the end of the buffer, then DAC_ERROR_RANGE is returned and no data is written.

It is possible to write all registers available in the hardware. The check for the current upper limit value of the buffer is not done. Therefore, it is possible to write data to the whole data buffer regardless of the current configuration.

DataArrayPtr has the fixed data type regardless of the current hardware or design time configuration and must always be used.

Return Value

- DAC_ERROR_OK — OK
- DAC_ERROR_RANGE — Parameter out of range

Example

The following example shows how to write to the DAC device buffer.

```
#define DAC_INTERNAL_BUFFER_SIZE     16
... variable definition section
static uint16_t DAC_BufferWaterMark = LDD_DAC_BUFFER_WATERMARK_L4;
static uint16_t *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);
```

20.4.10 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. */
    uint8_t             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
);
```

20.4.11 DAC_SetBufferMode()

This function sets the DAC internal buffer mode.

Synopsis

```
LDD_TError DAC_SetBufferMode(
    /* [IN] Device data structure pointer. */
    LDD_TDeviceDataPtr  DeviceData,
    /* [IN] - Buffer work mode. */
    LDD_DAC_TBufferMode Mode
);
```

Parameters

`DeviceData [in]` — Device data structure pointer.

`Mode [in]` — Buffer work mode.

Description

Selects the buffer work mode.

- LDD_DAC_BUFFER_DISABLED — Buffer Mode Disabled
- LDD_DAC_BUFFER_NORMAL_MODE — Buffer Normal Mode

  This is the default mode. The buffer works as a circular buffer. The read pointer increases by one every time when the trigger occurs. When the read pointer reaches the upper limit, it goes directly to zero in the next trigger event.
- LDD_DAC_BUFFER_SWING_MODE — Buffer Swing Mode

  This mode is similar to the Normal mode. However, when the read pointer reaches the upper limit, it does not go to the zero. It will descend by one in the next trigger event until zero is reached.
- LDD_DAC_BUFFER_OTSCAN_MODE — One-time scan mode

The read pointer increases by one every time the trigger occurs. When it reaches the upper limit, it stops. If the read pointer is reset to an address other than the upper limit, it will increase to the upper address and then stop.

Return Value

- DAC_ERROR_OK — OK

Example

The following example shows how to set the DAC buffer read pointer:

```
/* Set DAC internal buffer to circular mode */
DAC_Error = DAC_SetBufferMode(
        DAC_DevicePtr,
        LDD_DAC_BUFFER_NORMAL_MODE
);
```

20.4.12 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. */
    uint8_t             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
);
```

20.4.13 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. */
    uint8_t             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
);
```

20.4.14 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
);
```

20.5 Data Types Used by the DAC Driver API

This section describes the data types that are used by the DAC driver API.

20.5.1 LDD_TDeviceDataPtr

A pointer to a 32-bit unsigned integer and to the private structure containing component state information. Init method of the component creates the private state structure and returns the pointer to it. This pointer needs to be passed to every component method.

Definition

```
typedef void *LDD_TDeviceDataPtr;
```

20.5.2 LDD_RTOS_TDeviceDataPtr

A pointer to the structure used by RTOS containing driver-specific information. Init method receives this pointer and then passes it to all events and call-backs.

Definition

```
typedef struct
{
    /* DAC device number */
    uint8_t          DAC_DEVICE_NUMBER;
    /* DAC reference selection */
    uint8_t          DAC_REFSEL;
    /* DAC trigger mode */
    uint8_t          DAC_TRIGER_MODE;
    /* DAC buffer mode */
    uint8_t          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, * 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.

20.5.3 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;
```

20.5.4 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;
```

20.5.5 LDD_DAC_TData

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

Definition

```
typedef uint32_t LDD_DAC_TData;
```

20.5.6 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 uint32_t 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.

20.6 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.

20.7 Error Codes

The following error codes exist in the DAC device.

20.7.1 LDD_TError

Error identifier type specified in the *dac_ldd.h* header file. It is used to return error values.

Synopsis

```
typedef uint16_t 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 21
LWGPIO Driver

21.1 Overview

This section describes the Light-Weight GPIO (LWGPIO) driver that accompanies MQX RTOS. This driver is a common interface for GPIO modules.

The LWGPIO driver implements a custom API and does not follow the standard driver interface (I/O Subsystem). Therefore, it can be used before the I/O subsystem of MQX RTOS is initialized. LWGPIO driver is designed as a per-pin driver, meaning that an LWGPIO API call handles only one pin.

21.2 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.

21.3 Header Files

To use the LWGPIO driver, include the *lwgpio.h* header file and the platform specific header file, *lwgpio_mcf52xx.h*, into your application or into the BSP header file, *bsp.h*. The platform specific header file should be included before *lwgpio.h*.

The header file for Kinetis platforms is called *lwgpio_kgpio.h*.

21.4 API Function Reference

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

This function sets a property of the pin. For example a pull up resistor, a pull down resistor, drive strength, slew-rate, filters etc.

21.4.1 lwgpio_set_attribute ()

Synopsis

```
bool lwgpio_set_attribute
(
  LWGPIO_STRUCT_PTR  handle,
  uint32_t           attribute_id,
  uint32_t           value
)
```

Parameters

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

attribute_id [in] - Attribute identificator.

value [in] - Attribute value.

Description

MCUs have different properties for GPIO pins. These properties depend on the architecture and the GPIO or PORT module. This function handles these attributes. The attribute is defined by a special attribute ID. The value specifies requirements for the attribute (enable, disable, or a specific value). There are common attribute IDs and values placed in /io/lwgpio/lwgpio.h and driver specific attributes and values placed in /io/lwgpio/lwgpio_<driver>.h.

Return Value

- TRUE (success)
- FALSE (failure)

Example

The following example shows how to set the pull up for the button1 handle. This example returns FALSE if the pull up attribute is not available.

```
Lwgpio_set_attribute(&button1, LWGPIO_ATTR_PULL_UP, LWGPIO_AVAL_ENABLE);
```

21.4.2 lwgpio_init()

This function initializes the structure for a GPIO pin that will be used as a pin handle in the other API functions of the LWGPIO driver. It also performs basic GPIO register pre-initialization.

Synopsis

```
bool lwgpio_init
(
    LWGPIO_STRUCT_PTR handle,
    LWGPIO_PIN_ID     id,
    LWGPIO_DIR        dir,
    LWGPIO_VALUE      value
)
```

Parameters

handle [in/out] — Pointer to the LWGPIO_STRUCT structure that will be filled in.

id [in] — LWGPIO_PIN_ID number identifying pin (platform and peripheral specific).

dir [in] — LWGPIO_DIR enum value for initial direction control.

value [in] — LWGPIO_VALUE enum value for initial output control.

Description

The *lwgpio_init()* function has to be called prior to calling any other API function of the LWGPIO driver. This function initializes the LWGPIO_STRUCT structure. The pointer to the LWGPIO_STRUCT is passed as a *handle* parameter. To identify the pin, platform-specific LWGPIO_PIN_ID number is used.

The variable *dir* of type LWGPIO_DIR can have the following values:

- LWGPIO_DIR_INPUT - Presets pin into input state.
- LWGPIO_DIR_OUTPUT - Presets pin into output state.
- LWGPIO_DIR_NOCHANGE - Does not preset pin into input/output state.

The variable *value* of type LWGPIO_VALUE can have the following values:

- LWGPIO_VALUE_LOW - Presets pin into active low state.
- LWGPIO_VALUE_HIGH - Presets pin into active high state.
- LWGPIO_VALUE_NOCHANGE - Does not preset pin into low/high state.

If the *value* is set to LWGPIO_VALUE_LOW or LWGPIO_VALUE_HIGH and the *dir* parameter is passed as a LWGPIO_DIR_OUTPUT, the corresponding level is set on the GPIO output latch, if at all possible and depending on a peripheral, and the pin is set to the output state. This function does not configure the GPIO mode of the pin.

Return Value

- TRUE (Success)
- FALSE (Failure)

Example

The following example shows how to initialize the LWGPIO pin PTA-3 on MCF52259 MCU.

```
LWGPIO_STRUCT led1;
status = lwgpio_init(&led1,
                         LWGPIO_PORT_TA | LWGPIO_PIN3,
                         LWGPIO_DIR_OUTPUT,
                         LWGPIO_VALUE_HIGH);
if (status != TRUE)
{
        printf("Initializing GPIO as output failed.\n");
        _mqx_exit(-1);
}
```

21.4.3 lwgpio_set_functionality()

This function sets the functionality of the pin.

Synopsis

```
void lwgpio_set_functionality
(
    LWGPIO_STRUCT_PTR handle,
    uint32_t          functionality
)
```

Parameters

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

functionality [in] — An integer value which represents the requested functionality of the GPIO pin. This is a HW-dependent constant.

Description

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

Return Value

- None

Example

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

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

21.4.4 lwgpio_get_functionality()

This function gets the actual peripheral functionality of the pin. The pin peripheral function mode depends on the MCU.

Synopsis

```
uint32_t lwgpio_get_functionality
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

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

Description

This function is the inverse of the lwgpio_set_functionality(). It returns a value stored in the multiplexer register field which defines the desired functionality.

Return Value

- An integer value representing the actual pin functionality.

Example

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

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

21.4.5 lwgpio_set_direction()

This function sets direction (input or output) of the specified pin.

Synopsis

```
void lwgpio_set_direction
(
    LWGPIO_STRUCT_PTR handle,
    LWGPIO_DIR        dir
)
```

Parameters

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

dir [in] — One of the LWGPIO_DIR enum values.

Description

This function is used to change the direction of the specified pin. As this function does not change the pin's functionality, it is possible to set the direction of a pin that is currently not in the GPIO mode.

Return Value

- None

Example

The following example shows how to set the LWGPIO pin direction to the output on MCF52259.

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

21.4.6 lwgpio_set_value()

This function sets the pin state (low or high) of the specified pin.

Synopsis

```
void lwgpio_set_value
(
    LWGPIO_STRUCT_PTR handle,
    LWGPIO_VALUE      value
)
```

Parameters

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

value [in] — One of the LWGPIO_VALUE enum values.

Description

This function is used to change the specified pin state. As this function does not change either the pin's functionality or the direction, it is possible to set the pin state of a pin that is currently not in the GPIO mode. Similarly, it is possible to set the pin state of a pin that is set for input direction and have it ready for future changing of the pin direction.

Return Value

- None

Example

The following example shows how to set the pin state as "high" for the LWGPIO pin on MCF52259.

```
lwgpio_set_value(&led1, LWGPIO_VALUE_HIGH);
```

21.4.7 lwgpio_toggle_value()

This function toggles the pin state (low or high) of the specified pin.

Synopsis

```
void lwgpio_toggle_value
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

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

Description

This function is used for changing (toggling) the specified pin state.

Return Value

- none

Example

The following example shows how to toggle the pin state for the LWGPIO pin on MCF52259.

```
lwgpio_toggle_value(&led1);
```

21.4.8 lwgpio_get_value()

This function gets voltage value (low or high) of the specified pin.

Synopsis

```
LWGPIO_VALUE lwgpio_get_value
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

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

Description

This function is the inverse of the lwgpio_set_value() function. The direct relation between the physical pin state and the result of this function does not always exist, because this function gets the output buffer value rather than sampling pin voltage level of a pin that is set to output. To sample the pin voltage level, use lwgpio_get_raw() function. If the GPIO functionality is not assigned to the pin, the result of this function is not specified.

Return Value

- LWGPIO_VALUE - voltage value of the specified pin

Example

The following example shows how to get voltage level for the LWGPIO pin on MCF52259.

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

21.4.9 lwgpio_get_raw()

This function gets raw voltage value (low or high) of the specified pin if supported by target MCU.

Synopsis

```
LWGPIO_VALUE lwgpio_get_raw
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

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

Description

This function samples the pin signal to get the voltage value. If the GPIO functionality is not assigned to the pin, the result of this function is not specified.

Return Value

- LWGPIO_VALUE - Voltage value of the specified pin

Example

The following example shows how to get the physical voltage level for the LWGPIO pin on MCF52259.

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

21.4.10 lwgpio_int_init()

This function initializes interrupt for the specified pin.

Synopsis

```
bool lwgpio_int_init
(
    LWGPIO_STRUCT_PTR handle,
    LWGPIO_INT_MODE   mode
)
```

Parameters

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

mode [in] — Value consisting of a logical combination of the LWGPIO_INT_xxx flags.

Description

This function prepares the pin for the interrupt mode. It configures the interrupt peripheral to generate the interrupt flag. For most platforms, this function does not enable interrupts and it does not modify the GPIO peripheral settings. If there is a need to turn a

pin into a GPIO functionality in order to get the interrupt running, the user must do it manually prior to calling the lwgpio_int_init() function. In general, it is recommended to set the pin to the GPIO input state prior to the interrupt initialization.

Return Value

- TRUE (Success)
- FALSE (Failure)

Example

The following example shows how to initialize the rising edge interrupt for the LWGPIO pin PNQ.3 on MCF52259.

```
status = lwgpio_init(
    &btn_int,
    LWGPIO_PORT_NQ | LWGPIO_PIN3,
    LWGPIO_DIR_INPUT,
    LWGPIO_VALUE_NOCHANGE);
if (status == TRUE)
{
    status = lwgpio_int_init(&btn_int, LWGPIO_INT_MODE_RISING);
}
if (status != TRUE)
{
    printf("Initializing pin for interrupt failed.\n");
    _mqx_exit(-1);
}
```

21.4.11 lwgpio_int_enable()

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

Synopsis

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

Parameters

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

ena [in] — TRUE (enable), FALSE (disable).

Description

This function enables or disables interrupts for the specified pin (or set of pins- if so-called keyboard-interrupt peripheral is used) on the peripheral level. This effectively enables the interrupt channel from peripheral to the interrupt controller. This function does not set up interrupt controller to acknowledge interrupts. It is recommended to clear the flag with the lwgpio_int_clear_flag() function prior to the lwgpio_int_enable() function call.

Return Value

- None

Example

The following example shows how to enable the rising edge interrupt for the LWGPIO pin on MK40X256.

```
lwgpio_int_clear_flag(&btn_int);
lwgpio_int_enable(&btn_int, TRUE);
/* Enable interrupt for button on interrupt controller */
_bsp_int_init(lwgpio_get_int_vector(&btn_int), BUTTON_PRIORITY_LEVEL, 0, TRUE);
```

21.4.12 lwgpio_int_get_flag()

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

Synopsis

```
bool lwgpio_int_get_flag
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

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

Description

This function returns the pin interrupt flag on the peripheral. If the interrupt is so-called keyboard interrupt, it returns the interrupt flag for a set of pins.

Return Value

- TRUE if the flag is set
- FALSE if the flag is not set

Example

The following example checks the pending interrupt for the LWGPIO pin on MCF52259.

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

21.4.13 lwgpio_int_clear_flag()

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

Synopsis

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

Parameters

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

Description

This function clears the pin interrupt flag on the peripheral. If the interrupt is so-called keyboard interrupt, it clears the interrupt flag for a set of pins. This is typically called from the interrupt service routine, if the peripheral requires the flag being cleared by the software.

Return Value

- None

Example

The following example clears pending interrupt for the LWGPIO pin on MCF52259.

lwgpio_int_clear_flag(&btn_int);

21.4.14 lwgpio_int_get_vector()

This function gets the interrupt vector number that belongs to the pin or a set of pins.

Synopsis

```
uint32_t lwgpio_int_get_vector
(
    LWGPIO_STRUCT_PTR handle
)
```

Parameters

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

Description

This function returns the interrupt vector index for the specified pin. The obtained vector index can be used to install the interrupt by MQX RTOS.

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.

```
uint32_t vector = lwgpio_int_get_vector(&btn1);
 _int_install_isr(vector, int_callback, (void *) param);
```

21.5 Macro Functions Exported by the LWGPIO Driver

LWGPIO driver exports inline functions (macros) for an easy pin driving without needing to use the pin handle structure. The structure is initiated internally in the inline code. These functions are available for every platform and are generic. They are defined in the *lwgpio.h* file.

21.5.1 lwgpio_set_pin_output()

This macro puts the specified pin into the output state with the defined output value.

Synopsis

```
bool inline lwgpio_set_pin_output(
    LWGPIO_PIN_ID id,
    LWGPIO_VALUE pin_state
)
```

Parameters

id [in] — LWGPIO_PIN_ID number identifying pin which is platform and peripheral specific.

pin_state [in] — LWGPIO_VALUE enum value for initial output control.

Description

This inline function switches the specified pin into the output state. The output level is defined by the *pin_state* parameter.

Return Value

- TRUE (success)
- FALSE (failure)

Example

The following example shows how to set high voltage level output for the LWGPIO pin PTA.3 on MCF52259.

```
lwgpio_set_pin_output(LWGPIO_PORT_TA | LWGPIO_PIN3, LWGPIO_VALUE_HIGH);
```

21.5.2 lwgpio_toggle_pin_output()

This macro changes (toggles) the output value of the specified pin and requires the pin multiplexer to be set to the GPIO function. Otherwise, the pin output is not going to change.

Synopsis

```
bool inline lwgpio_toggle_pin_output(
    LWGPIO_PIN_ID id
)
```

Parameters

id [in] — LWGPIO_PIN_ID number identifying pin which is platform and peripheral specific.

Description

This inline function switches the specified pin into the output state and toggles the output value. The output level is taken from the output buffer value.

Return Value

- TRUE (success)
- FALSE (failure)

Example

The following example shows how to toggle output for the LWGPIO pin PTA.3 on MCF52259.

```
lwgpio_toggle_pin_output(LWGPIO_PORT_TA | LWGPIO_PIN3);
```

21.5.3 lwgpio_get_pin_input()

This function gets voltage value (low or high) of the specified pin.

Synopsis

```
LWGPIO_VALUE inline lwgpio_get_pin_input
(
    LWGPIO_STRUCT_PTR id
)
```

Parameters

id [in] — LWGPIO_PIN_ID number identifying pin which is platform and peripheral specific.

Description

This function gets the input voltage level value in the same way as lwgpio_get_value() function does.

Return Value

- LWGPIO_VALUE_HIGH - Voltage value of specified pin is high
- LWGPIO_VALUE_LOW - Voltage value of specified pin is low
- LWGPIO_VALUE_NOCHANGE - Could not configure pin for input (failure)

Example

The following example shows how to get (pre-set) voltage level for the LWGPIO pin PTA.3 on MCF52259.

```
value = lwgpio_get_pin_input(LWGPIO_PORT_TA | LWGPIO_PIN3);
if (value == LWGPIO_VALUE_NOCHANGE)
{
    printf("Can not configure pin PTA.3 for input.\n");
    _mqx_exit(-1);
}
```

21.6 Data Types Used by the LWGPIO API

The following data types are used within the LWGPIO driver.

21.6.1 LWGPIO_PIN_ID

This 32 bit number specifies the pin on the MCU. The number is MCU-specific.

```
typedef uint32_t LWGPIO_PIN_ID;
```

In general, LWGPIO_PIN_ID value consists of two logically OR-ed constants: port value and pin value. Both of these macro values have a common nomenclature across all platforms:

```
LWGPIO_PIN_ID pin_id = LWGPIO_PORT_xyz | LWGPIO_PIN_z;
```

Though these macros have common format and style, they are MCU-specific. Every MCU or platform has its own macros defined. The constants can be found in the *lwgpio_<mcu>.h* file and should be used to create the LWGPIO_PIN_ID value.

21.6.2 LWGPIO_STRUCT

A pointer to this structure is used as a handle for the LWGPIO driver API functions. The content of this structure is MCU-specific. This structure has to be allocated in the user application space such as heap and stack before calling lwgpio_init() function.

21.6.3 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.

21.6.4 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 cannot be obtained.

21.6.5 LWGPIO_INT_MODE

This integer value specifies the interrupt mode of the pin. The value is generic.

```
typedef unsigned char LWGPIO_INT_MODE;
```

In general, LWGPIO_INT_MODE value consists of several logically OR-ed constants. The same macro can have a different value on a different MCU.

```
LWGPIO_INT_MODE_RISING
LWGPIO_INT_MODE_FALLING
LWGPIO_INT_MODE_HIGH
LWGPIO_INT_MODE_LOW
```

Note that although these macros are MCU defined, it does not mean that MCU supports any combination. In case of an unsupported combination, the function with incorrect LWGPIO_INT_MODE will return the failure status.

21.7 Example

The example for the LWGPIO driver that shows how to use LWGPIO driver API functions is provided with the MQX RTOS installation and it is located in `mqx/examples/lwgpio` directory.

Chapter 22
Low Power Manager

22.1.1 Overview

This section describes the Low Power Manager (LPM) that accompanies the MQX RTOS. The Freescale MQX RTOS provides low power functionality in terms of run-time clock frequency changes and CPU/peripheral operation mode changes such as shutting down peripheral clocks, module enabling, and setting pin multiplexer. The feature of CPU core sleep is also available in the idle task designed for further power saving while all user tasks are blocked.

LPM is a common interface that enables the user application to switch between pre-defined low power operation modes and clock configurations in a controlled way at runtime. A user defines the behavior of the CPU core and selected low power-enabled peripheral drivers in each operation mode. Regarding the clock configurations, LPM serves as a wrapper around another MQX component, the Clock Manager.

The purpose of LPM is to gather all information needed for low power system change and manage the preparation and recovery phases with a few function calls. Drivers, stacks, and other user state machines which are affected by different low power settings register their callback handlers at the LPM. These handlers are used by the LPM to notify all registered drivers before any operation mode change and before and after the clock configuration change. The drivers have to adapt to the new global settings within the callbacks according to their behavior specified for the operation mode and clock configuration. All registered drivers are accessed by the LPM in a user-defined order called a "dependency level".

LPM implements the custom C language API and does not follow the standard POSIX-like driver interface (I/O Subsystem).

LPM functionality is currently available for Kinetis platforms only and must be explicitly enabled in *user_config.h* using the MQX_ENABLE_LOW_POWER configuration option.

The system timer and serial driver are currently the only low power-enabled drivers. For more information about low power mode implementation in a particular driver, see "Low Power Support" chapter in the corresponding driver section.

22.1.2 Source Code Location

LPM module is a part of the BSP library since the behavior is related to a particular board and to peripheral drivers. There are low power definitions and behavior structures defined for the CPU core *(init_lpm.h, init_lpm.c)* and for supported drivers *(init_sci.c)* in the BSP directory.

The source code files for the LPM are located in source/io/lpm directory. The *lpm_* file prefix is used for all LPM module related API files. Additional functionality is also added to the source code for all low power-enabled drivers.

22.1.3 Header Files

To use the LPM functionality, include the *bsp.h* header file into your application. It already contains all needed header file and includes all definitions.

22.1.4 API Function Reference

This section serves as a function reference for the LPM module.

22.1.4.1 _lpm_install()

The function installs and enables the Low Power Manager within MQX software.

Synopsis

```
_mqx_uint _lpm_install
(
const LPM_CPU_OPERATION_MODE  *operation_modes,
LPM_OPERATION_MODE             default_mode
)
```

Parameters

operation_modes [in] - Pointer of the CPU core operation modes array.

default_mode [in] - Enumerated value of default (current) operation mode.

Description

This function installs the MQX LPM with given CPU core behavior specification in all operation modes and with default (currently running) operation mode. Driver registrations and power mode switching is possible after the successful return from this function.

By default, this function is called in the BSP startup code before any driver installation takes place, so the user application shouldn't call this function again.

Return Value

- MQX_INVALID_PARAMETER - Wrong parameter
- MQX_COMPONENT_EXISTS - LPM already installed
- MQX_IO_OPERATION_NOT_AVAILABLE - Possible memory problem
- MQX_OK - Success

Example

The following example shows the installation of the LPM:

```
if (MQX_OK != _lpm_install (LPM_CPU_OPERATION_MODES, LPM_OPERATION_MODE_RUN))
{
    printf ("Error during LPM install!\n");
}
```

22.1.4.2 _lpm_uninstall()

The function uninstalls LPM functionality from MQX software.

Synopsis

```
_mqx_uint _lpm_uninstall
(
    void
)
```

Parameters

None.

Description

This function uninstalls the LPM from MQX RTOS leaving current operation mode and clock configuration unchanged. No more LPM function calls may occur in the application after this function is called.

Return Value

- MQX_IO_OPERATION_NOT_AVAILABLE - LPM not installed or memory problem
- MQX_OK - Success

22.1.4.3 _lpm_register_driver()

This function registers a driver that must be notified about low power system changes at LPM.

Synopsis

```
_mqx_uint _lpm_register_driver
(
const LPM_REGISTRATION_STRUCT_PTR  driver_registration_ptr,
const void                        *driver_specific_data_ptr,
_mqx_uint                         *registration_handle_ptr
)
```

Parameters

driver_registration_ptr [in] - Pointer to a registration structure with driver callbacks.

driver_specific_data_ptr [in] - Pointer to a driver specific data to be passed to callbacks.

registration_handle_ptr [out] - Unique driver registration handle.

Description

This function registers notification callbacks of the driver for operation mode changes and for clock configuration changes. For operation mode changes, the corresponding callback is called before the actual mode change. For clock configuration changes, the corresponding callback is called both before and after the change is made. The callback routines have unified API as described below.

The callback handler can return an error which means that the driver is not ready or able to switch to a given low power mode. In this case, no low power system change is made and a rollback takes place. The rollback is done by notifying all drivers already processed (except for the one whose callback caused an error) in reverse order and with the original mode specified.

Besides callbacks, there is also a dependency level specified for each driver during registration. The dependency level affects the order in which the registered drivers are notified about low power system changes. For pre-change notifications, the lower

dependency level drivers are processed first. The order of registration is used for drivers at the same dependency level. For post- notifications or for rollback notifications in case of failure, the order is reversed.

If the driver registration succeeds, the function returns MQX_OK and unique driver registration handle that must be used later during the driver unregister process.

Low power-enabled POSIX drivers in MQX RTOS register themselves at the LPM automatically during their installation, so user application shouldn't register them explicitly again.

Return Value

- MQX_INVALID_PARAMETER - Wrong parameter
- MQX_IO_OPERATION_NOT_AVAILABLE - LPM not installed
- MQX_OUT_OF_MEMORY - Possible memory problem
- MQX_OK - Success

Example

The following example shows the automatic registration of polled serial driver into LPM:

```
IO_SERIAL_POLLED_DEVICE_STRUCT_PTR  dev_ptr;
LPM_REGISTRATION_STRUCT             registration;
registration.CLOCK_CONFIGURATION_CALLBACK =
    _io_serial_polled_clock_configuration_callback;
registration.OPERATION_MODE_CALLBACK      =
    _io_serial_polled_operation_mode_callback;
registration.DEPENDENCY_LEVEL             =
    BSP_LPM_DEPENDENCY_LEVEL_SERIAL_POLLED;
result = _lpm_register_driver
            (
                &registration, dev_ptr,
                &(dev_ptr->LPM_INFO.REGISTRATION_HANDLE)
            );
if (MQX_OK == result)
{
    _lwsem_create (&(dev_ptr->LPM_INFO.LOCK), 1);
    dev_ptr->LPM_INFO.FLAGS = 0;
}
```

22.1.4.4 _lpm_unregister_driver()

The function unregisters driver from the LPM.

Synopsis

```
_mqx_uint _lpm_unregister_driver
(
    _mqx_uint registration_handle
)
```

Parameters

registration_handle [in] - Unique driver registration handle.

Description

This function unregisters a driver from the LPM by using the unique handle returned by registration function. If the function succeeds, the driver keeps current low power settings and it's not notified anymore about low power system changes.

Return Value

- MQX_INVALID_PARAMETER - Wrong parameter
- MQX_IO_OPERATION_NOT_AVAILABLE - LPM not installed
- MQX_INVALID_HANDLE - Corresponding driver not registered at LPM
- MQX_OK - Success

Example

The following example shows the automatic unregister process of polled serial driver from LPM:

```
IO_SERIAL_POLLED_DEVICE_STRUCT_PTR dev_ptr;
_lpm_unregister_driver (dev_ptr->LPM_INFO.REGISTRATION_HANDLE);
_lwsem_destroy (&(dev_ptr->LPM_INFO.LOCK));
```

22.1.4.5 _lpm_set_clock_configuration()

The function switches the system to a given low power clock configuration including all preparation and recovery actions.

Synopsis

```
_mqx_uint _lpm_set_clock_configuration
(
    BSP_CLOCK_CONFIGURATION clock_configuration
)
```

Parameters

clock_configuration [in] - Clock configuration identifier defined in the BSP.

Description

This function notifies all registered drivers about the clock configuration that is to be switched to. The pre- notifications are made in ascending order according to the driver dependency level or according to the order of registration in case of the same dependency level. After pre-notifications, this function changes the clock configuration physically and post-notifies all drivers in reverse order that the clock configuration has been changed.

In case of any failure or error reported by any driver during the pre-notifications phase, the clock configuration is not changed and all drivers already processed are notified again in reverse order with the original clock information.

Return Value

- MQX_INVALID_PARAMETER - Wrong parameter
- MQX_IO_OPERATION_NOT_AVAILABLE - LPM not installed or clock configuration change failed
- MQX_OK - Success

Example

The following example shows the clock configuration change:

```
if (MQX_OK != _lpm_set_clock_configuration(BSP_CLOCK_CONFIGURATION_1))
{
       printf ("Clock configuration not changed!\n");
}
```

22.1.4.6 _lpm_get_clock_configuration()

The function returns the identifier for the clock configuration in which MQX RTOS is currently running.

Synopsis

```
BSP_CLOCK_CONFIGURATION _lpm_get_clock_configuration
(
   void
)
```

Parameters

None

Description

This function returns active clock configuration or -1 if the LPM is not installed.

Return Value

- BSP_CLOCK_CONFIGURATION - One of the predefined enumerated values
- -1 - When LPM is not installed

Example

The following example shows acquiring the current clock configuration:

```
clock_configuration = _lpm_get_clock_configuration();
```

22.1.4.7 _lpm_set_operation_mode()

The function switches the system to given low power operation mode and performs all preparation and recovery actions.

Synopsis

```
_mqx_uint _lpm_set_operation_mode
(
    LPM_OPERATION_MODE operation_mode
)
```

Parameters

operation_mode [in] - Operation mode identifier defined in the BSP.

Description

This function pre-notifies all registered drivers about the operation mode that is to be switched to. The notifications are made in ascending order according to the driver dependency level or according to the order of registration in case of the same dependency level. After the notifications, this function actually changes the power mode of the CPU core.

In case of any failure or error reported by any driver during the pre-notifications phase, the power mode of the CPU core is not changed and all already processed drivers are notified in reverse order with the original operation mode parameter.

Note

This function may block the CPU core and may not return until specified wakeup event occurs. It may also restart the idle task to enable/disable the idle task sleep feature.

Return Value

- MQX_INVALID_PARAMETER - Wrong parameter
- MQX_INVALID_CONFIGURATION - Wrong CPU core operation mode settings

- MQX_IO_OPERATION_NOT_AVAILABLE - LPM not installed or operation mode change failed
- MQX_OK - Success

Example

The following example shows the operation mode change:

```
if (MQX_OK != _lpm_set_operation_mode(LPM_OPERATION_MODE_WAIT))
{
    printf ("Operation mode not changed!\n");
}
```

22.1.4.8 _lpm_get_operation_mode ()

The function returns identifier of the operation mode in which MQX RTOS is currently running.

Synopsis

```
LPM_OPERATION_MODE _lpm_get_operation_mode
(
   void
)
```

Parameters

None.

Description

This function returns active operation mode or -1 if the LPM is not installed.

Return Value

- LPM_OPERATION_MODE - One of the predefined enumerated values
- -1 - When LPM is not installed

Example

The following example shows acquiring the current operation mode:

```
operation_mode = _lpm_get_operation_mode();
```

22.1.4.9 _lpm_wakeup_core()

This platform-specific function signals the CPU core not to return to sleep mode again after the ISR finishes. This function should be called from the ISR.

Synopsis

```
void _lpm_wakeup_core
(
     void
)
```

Parameters

None.

Description

One of the possible low power operation modes is "execute interrupts only", so the CPU core has no chance to exit this mode without cooperation from the interrupt service routine. To pass control back to the tasks, the application must use this function within an interrupt routine to keep the CPU running after the ISR finishes.

This function is currently available for Kinetis platform only. It clears the SLEEPONEXIT flag in the Kinetis system control register.

Return Value

None.

Example

The following example shows how to keep the CPU core awake after ISR:

```
void ISR (void *data)
{
...
    if (time_to_let_the_tasks_run)
    {
       _lpm_wakeup_core();
    }
}
```

22.1.4.10 _lpm_idle_sleep_setup()

This function enables/disables the feature of the CPU core sleep in idle task. It may also restart the idle task.

Synopsis

```
void _lpm_idle_sleep_setup
(
    bool enable
)
```

Parameters

enable [in] - Enable or disable the feature.

Description

The function enables or disables the feature CPU core sleep during the execution of the idle task. By default CPU core sleep during idle task is turned off. For that purpose, the function may restart the idle task. When enabled, the idle task sleep feature is relevant only in the CPU power modes RUN, WAIT, VLPR, and VLPW.

Return Value

None.

Example

The following example shows how to enable idle task sleep feature:

```
_lpm_idle_sleep_setup(TRUE);
```

22.1.4.11 _lpm_idle_sleep_check()

This function checks whether it's possible to perform idle task sleep in the current operation mode.

Synopsis

```
bool _lpm_idle_sleep_check
(
     void
)
```

Parameters

None.

Description

The function checks current settings of the power mode registers to find out whether it is possible to put CPU core to sleep in the idle task. This function is used internally by the LPM and is available for information purposes only.

Return Value

- TRUE - Current settings allow to execute core sleep in idle task.
- FALSE - Otherwise.

Example

The following example shows how to check idle sleep feature availability in the current operation mode:

```
sleep_allowed = _lpm_idle_sleep_check();
```

22.1.4.12 driver_notification_callback()

All driver notification functions must be defined with the following unified type.

Synopsis

```
LPM_NOTIFICATION_RESULT driver_notification_callback
(
LPM_NOTIFICATION_STRUCT_PTR notification_ptr,
void                        *driver_specific_data_ptr
)
```

Parameters

notification_ptr [in] - Notification type, target operation mode, and clock configuration.

driver_specific_data_ptr [in] - Pointer to the driver specific data that was passed to the low power manager during driver registration.

Description

Notification callback should change hardware settings of the driver according to the given low power identifiers and driver specific behavior structures that are part of its initialization information. Notification callback can return an error to indicate that the low power change cannot be fulfilled. This causes a rollback during a particular operation mode change or clock configuration change.

There can be a significant time delay between pre-notification and post-notification. Therefore, locking the access to the driver in the meantime may be necessary to avoid any unexpected behavior.

Return Value

- LPM_NOTIFICATION_RESULT_OK - Success
- LPM_NOTIFICATION_RESULT_ERROR - Driver not ready or able to switch to given low power settings

Example

The following example shows possible serial clock configuration callback:

```
LPM_NOTIFICATION_RESULT serial_clock_configuration_callback
(
    /* [IN] Low power notification */
    LPM_NOTIFICATION_STRUCT_PTR notification_ptr,
```

```
    /* [IN/OUT] Device specific data */
    void                        *driver_specific_data_ptr
)
{
    if (LPM_NOTIFICATION_TYPE_PRE == notification->NOTIFICATION_TYPE)
    {
        if (   (BSP_CLOCK_CONFIGURATION_2MHZ == notification->CLOCK_CONFIGURATION)
            && (LPM_OPERATION_MODE_RUN == notification->OPERATION_MODE))
        {
            /* Unable to operate when on 2MHZ */
            return LPM_NOTIFICATION_RESULT_ERROR;
        }
    }
    if (LPM_NOTIFICATION_TYPE_POST == notification->NOTIFICATION_TYPE)
    {
        SERIAL_INIT_STRUCT_PTR  init;
        uint32_t input_clock;
        init = ((SERIAL_STRUCT_PTR)driver_specific_data_ptr)->DEV_INIT_DATA_PTR;
        input_clock = cm_get_clock
                        (
                            notification->CLOCK_CONFIGURATION,
                            init->CLOCK_SOURCE
                        );
        serial_set_baudrate
                        (
                            driver_specific_data_ptr,
                            input_clock,
                            init->BAUD_RATE
                        );
    }
    return LPM_NOTIFICATION_RESULT_OK;
}
```

22.1.4.13 _lpm_register_wakeup_callback

This function is used to install ISR for LLUW interrupt and register user wakeup callback function executed by LLWU ISR.

Synopsis

```
    (
        uint32_t llwu_vector,
        uint8_t llwu_prior,
        void (* llwu_user_wakeup_callback)(uint32_t)
    )
```

Parameters

```
llwu_vector [in] — LLWU vector number
llwu_prior [in] — priority of LLWU interrupt
llwu_user_wakeup_callback [in] — user callback function
```

Description

This function installs LLWU interrupt ISR and registers the interrupt with input priority value.

Return Value

None

22.1.4.14 _lpm_unregister_wakeup_callback

This function is used to uninstall ISR for LLUW interrupt.

Synopsis

```
(
    uint32_t llwu_vector
)
```

Parameters

`llwu_vector [in] — LLWU vector number`

Description

This function uninstalls LLWU interrupt ISR and unregisters the interrupt with core.

Return Value

None

22.1.4.15 _lpm_llwu_isr

This function is ISR of LLWU interrupt.

Synopsis

```
(
    void *llwu_param_ptr
)
```

Parameters

`llwu_param_ptr [in] — LLWU ISR parameter`

Description

This function will clear flags of LLWU interrupt then execute user callback function registered.

Return Value

None

22.1.4.16 _lpm_llwu_clear_flag

This function clear LLWU flags then return to the caller.

Synopsis

```
(
    uint32_t *llwu_fx_ptr
)
```

Parameters

llwu_fx_ptr [out] — Pointer stores value of LLWU flags

Description

This function is specific for KINETIS and CF+. It's implemented in lpm_kinetis.c and lpm_cfplus.c.

Return Value

None

22.1.4.17 _lpm_get_reset_source

This function is used to get RESET source of core after wake up.

Synopsis

```
(
    void
)
```

Parameters

None

Description

Depends on the family chip; this function is implemented in lpm_mc.c or lpm_smc.c and return different RESET sources.

Return Value

- MQX_RESET_SOURCE_LLWU
- MQX_RESET_SOURCE_LOW_VOLTAGE_DETECT
- MQX_RESET_SOURCE_LOSS_OF_CLOCK
- MQX_RESET_SOURCE_WATCHDOG
- MQX_RESET_SOURCE_EXTERNAL_PIN
- MQX_RESET_SOURCE_POWER_ON

- MQX_RESET_SOURCE_JTAG
- MQX_RESET_SOURCE_CORE_LOCKUP
- MQX_RESET_SOURCE_SOFTWARE
- MQX_RESET_SOURCE_MDM_AP
- MQX_RESET_SOURCE_EZPT
- MQX_RESET_SOURCE_TAMPER
- MQX_RESET_SOURCE_INVALID

22.1.4.18 _lpm_get_wakeup_source

This function is used to get WAKEUP source of core after wake up.

Synopsis

```
(
    void
)
```

Parameters

None

Description

Depend on family chip; this function is implemented in lpm_mc.c or lpm_smc.c and return different WAKEUP sources.

Return Value

- MQX_WAKEUP_SOURCE_LLWU_P0
- MQX_WAKEUP_SOURCE_LLWU_P1
- MQX_WAKEUP_SOURCE_LLWU_P2
- MQX_WAKEUP_SOURCE_LLWU_P3
- MQX_WAKEUP_SOURCE_LLWU_P4
- MQX_WAKEUP_SOURCE_LLWU_P5
- MQX_WAKEUP_SOURCE_LLWU_P6
- MQX_WAKEUP_SOURCE_LLWU_P7

- MQX_WAKEUP_SOURCE_LLWU_P8
- MQX_WAKEUP_SOURCE_LLWU_P9
- MQX_WAKEUP_SOURCE_LLWU_P10
- MQX_WAKEUP_SOURCE_LLWU_P11
- MQX_WAKEUP_SOURCE_LLWU_P12
- MQX_WAKEUP_SOURCE_LLWU_P13
- MQX_WAKEUP_SOURCE_LLWU_P14
- MQX_WAKEUP_SOURCE_LLWU_P15
- MQX_WAKEUP_SOURCE_MODULE0
- MQX_WAKEUP_SOURCE_MODULE1
- MQX_WAKEUP_SOURCE_MODULE2
- MQX_WAKEUP_SOURCE_MODULE3
- MQX_WAKEUP_SOURCE_MODULE4
- MQX_WAKEUP_SOURCE_MODULE5
- MQX_WAKEUP_SOURCE_MODULE6
- MQX_WAKEUP_SOURCE_MODULE7
- MQX_WAKEUP_SOURCE_INVALID

22.1.4.19 _lpm_write_rfvbat

This function is used to write data to a specific RFVBAT register file

Synopsis

```
(
    uint8_t channel,
    uint32_t data
)
```

Parameters

```
channel  [in] — Channel of RFVBAT register
data [in] — Data is going to be written to the RFVBAT register file
```

Description

Data in RFVBAT register files is not be erased in all power modes and is only reset during power-on reset.

Return Value

- MQX_INVALID_PARAMETER
- MQX_OK

22.1.4.20 _lpm_read_rfvbat

This function is used to read data from a specific RFVBAT register file

Synopsis

```
(
    uint8_t channel,
    uint32_t *data_ptr
)
```

Parameters

```
channel  [in] — Channel of RFVBAT register
data_ptr [out] — Buffer stores content of the RFVBAT register file
```

Description

Data in RFVBAT register files is not be erased in all power modes and is only reset during power-on reset.

Return Value

- MQX_INVALID_PARAMETER
- MQX_OK

22.1.4.21 _lpm_write_rfsys

This function is used to write data to a specific RFSYS register file

Synopsis

```
(
    uint8_t channel,
    uint32_t data
)
```

Parameters

```
channel  [in] — Channel of RFSYS register
data [in] — Data is going to be written to the RFSYS register file
```

Description

Data in RFSYS register files is not be erased in all power modes and is only reset during power-on reset.

Return Value

- MQX_INVALID_PARAMETER
- MQX_OK

22.1.4.22 _lpm_read_rfsys

This function is used to read data from a specific RFSYS register file

Synopsis

```
    (
        uint8_t channel,
        uint32_t *data_ptr
    )
```

Parameters

```
channel  [in] — Channel of RFSYS register
data_ptr [out] — Buffer stores content of the RFSYS register file
```

Description

Data in RFSYS register files is not be erased in all power modes and is only reset during power-on reset.

Return Value

- MQX_INVALID_PARAMETER
- MQX_OK

22.1.5 Remarks

The points to consider when working with LPM in MQX RTOS:

- All functions described above are thread-safe and should not be called from an ISR, except in the explicitly stated cases. These functions should also not be called from the notification callbacks of registered drivers. In particular, the idle task sleep feature cannot be switched from within any ISR.
- The LPM is automatically installed during BSP initialization before any other driver installation, so it should not be done again by the application.
- All low power-enabled POSIX drivers register/unregister themselves automatically during their installation/uninstallation, so it shouldn't be done again by the application.
- The Kinetis BAT modes are not currently supported. In the *init_lpm.c* file, mapping exists between generic operation mode identifiers and the supported Kinetis CPU core power modes. This mapping can be changed by the user. All available Kinetis operation modes can be found in the *lpm_kinetis.h* file.
- After wakeup from an operation mode where CPU is inactive, the system remains in the last operation mode set. It is an application's responsibility to switch to another mode immediately after wakeup by calling *_lpm_set_operation_mode()* function.
- Idle sleep feature may cause problems while debugging. It is recommended to turn the feature off for debug purposes. The feature is disabled by default and as long as the LPM is not installed.

22.1.6 Data Types Used by the LPM Driver API

The following data types are defined regarding the LPM functionality.

22.1.6.1 LPM_OPERATION_MODE

This enumerated type defines identifiers of the generic operation modes available in the BSP and their overall count. Arrays of structures are defined which describe the behavior of the CPU core and each low power-enabled peripheral for all of the following operation modes in the BSP.

```
typedef enum
{
    LPM_OPERATION_MODE_RUN = 0,
    LPM_OPERATION_MODE_WAIT,
    LPM_OPERATION_MODE_SLEEP,
    LPM_OPERATION_MODE_STOP,
    LPM_OPERATION_MODES
} LPM_OPERATION_MODE;
```

22.1.6.2 LPM_NOTIFICATION_TYPE

This enumerated type specifies whether the driver notification is done before or after the actual low power system change. It is passed to all notification callbacks.

```
typedef enum {
    LPM_NOTIFICATION_TYPE_PRE,
    LPM_NOTIFICATION_TYPE_POST
} LPM_NOTIFICATION_TYPE;
```

22.1.6.3 LPM_NOTIFICATION_RESULT

One of these enumerated values should be returned by any notification callback. When LPM_NOTIFICATION_RESULT_ERROR is returned, it forces LPM not to make low power system changes and to rollback all already processed drivers to the previous mode.

```
typedef enum {
    LPM_NOTIFICATION_RESULT_OK,
    LPM_NOTIFICATION_RESULT_ERROR
} LPM_NOTIFICATION_RESULT;
```

22.1.6.4 LPM_NOTIFICATION_STRUCT

A pointer to this structure is passed to all notification callback handlers to inform them about a type of notification and about low power settings to be switched to.

```
typedef struct lpm_notification_struct {
    LPM_NOTIFICATION_TYPE   NOTIFICATION_TYPE;
    LPM_OPERATION_MODE      OPERATION_MODE;
    BSP_CLOCK_CONFIGURATION CLOCK_CONFIGURATION;
} LPM_NOTIFICATION_STRUCT, * LPM_NOTIFICATION_STRUCT_PTR;
```

22.1.6.5 LPM_REGISTRATION_STRUCT

This structure has to be filled and passed during the driver registration at the LPM. It specifies both the operation mode and clock configuration callbacks and the order of processing among other registered drivers. One of the callbacks can also be NULL if not required. See example below:

```
typedef struct lpm_registration_struct {
    LPM_NOTIFICATION_CALLBACK CLOCK_CONFIGURATION_CALLBACK;
    LPM_NOTIFICATION_CALLBACK OPERATION_MODE_CALLBACK;
    _mqx_uint                 DEPENDENCY_LEVEL;
} LPM_REGISTRATION_STRUCT, * LPM_REGISTRATION_STRUCT_PTR;
```

22.1.7 Platform-Specific Data Types Used by the LPM API

The following data types are used in generic LPM API calls, but are defined differently on each processor platform. So far only the Kinetis platform is supported by the MQX LPM.

22.1.7.1 LPM_CPU_POWER_MODE_INDEX

The enumerated type defines identifiers for all supported Kinetis specific CPU core power modes and their overall count.

```
typedef enum
{
    LPM_CPU_POWER_MODE_RUN = 0,
    LPM_CPU_POWER_MODE_WAIT,
    LPM_CPU_POWER_MODE_STOP,
    LPM_CPU_POWER_MODE_VLPR,
    LPM_CPU_POWER_MODE_VLPW,
    LPM_CPU_POWER_MODE_VLPS,
    LPM_CPU_POWER_MODE_LLS,
    LPM_CPU_POWER_MODES
} LPM_CPU_POWER_MODE_INDEX;
```

VLLSx

```
typedef enum
{
   LPM_CPU_POWER_MODE_RUN = 0,
   LPM_CPU_POWER_MODE_WAIT,
   LPM_CPU_POWER_MODE_STOP,
   LPM_CPU_POWER_MODE_VLPR,
   LPM_CPU_POWER_MODE_VLPW,
   LPM_CPU_POWER_MODE_VLPS,
   LPM_CPU_POWER_MODE_LLS,
   LPM_CPU_POWER_MODE_VLLS3,
   LPM_CPU_POWER_MODE_VLLS2,
   LPM_CPU_POWER_MODE_VLLS1,
   LPM_CPU_POWER_MODES
   } LPM_CPU_POWER_MODE_INDEX;
```

22.1.7.2 LPM_CPU_OPERATION_MODE

The platform-specific structure describes the behavior of the CPU core in one of the operation modes available. It maps from one of the generic operation modes to one of the Kinetis-specific CPU core power modes. Additional operation mode flag can be specified here. Also wakeup settings of LLWU registers can be specified here (applies only for Kinetis LLS mode).

```
typedef struct lpm_cpu_operation_mode {
    LPM_CPU_POWER_MODE_INDEX MODE_INDEX;
    uint8_t                  FLAGS;
    uint8_t                  PE1;
```

```
    uint8_t                         PE2;
    uint8_t                         PE3;
    uint8_t                         PE4;
    uint8_t                         ME;
} LPM_CPU_OPERATION_MODE, * LPM_CPU_OPERATION_MODE_PTR;
```

Table 22-1. LPM_CPU_OPERATION_MODE Flags

FLAGS	Description
LPM_CPU_POWER_MODE_FLAG_SLEEP_ON_EXIT	This flag tells the LPM that specified operation mode is "execute interrupts only", i.e., the CPU core is active only for the interrupt service routines and it returns to sleep upon exit out of any ISR. Applies only for Kinetis wait and stop modes. To pass the control back to tasks, function *_lpm_wakeup_core()* must be called within any ISR.

22.1.8 Example

The example of the low power feature that shows how to use LPM API functions is provided together with the MQX installation and is located in mqx/examples/lowpower directory.

The default settings of operation modes, clock configurations, and behavior definitions for the CPU core and low power enabled drivers can be found in the corresponding BSP directory.

Chapter 23
Resistive Touch Screen Driver

23.1 Overview

This chapter describes the device driver which is a common interface for four-wire resistive touch screens as shown on -.

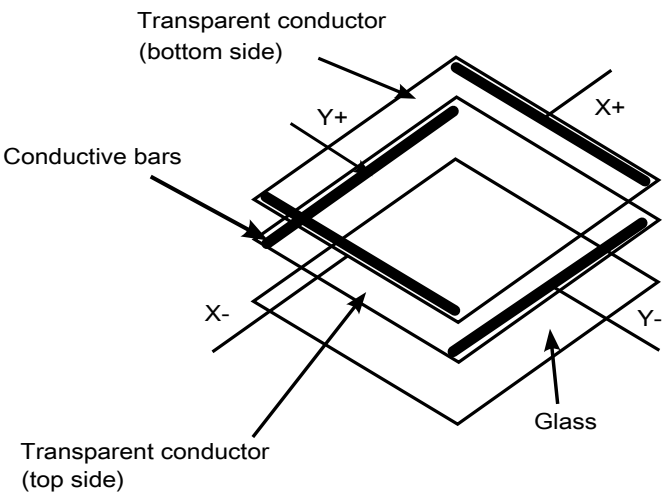

Figure 23-1. Four-wire Resistive Touch Screen

The touch screen driver uses Lightweight GPIO driver to toggle voltage on depicted electrodes and MQX ADC driver for measuring voltage on electrodes marked as X+ and Y+.

The x and y coordinates of a touch are read in two steps as described below:

- Before the measurement all electrodes are grounded, set to low using LWGPIO driver, to discharge electrodes.

- Measuring of X coordinate - Before the measurement X+ is driven to high (Ucc), the X- grounded and Y- set to high impedance using LWGPIO driver. The position is measured on Y+ electrode using the ADC driver.
- When a touch is detected, the electrodes are grounded again and the measurement continues analogically to measure Y coordinate.

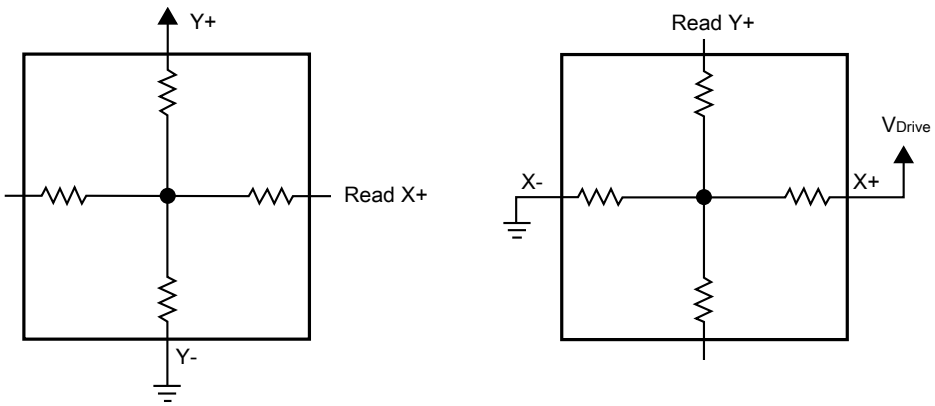

Figure 23-2. Measuring of Voltage on Electrodes X+ and Y+

For touch screen functionality, it is crucial to wire electrodes with MCU. The X+ and Y+ pins must offer both GPIO and ADC functionality for both. It is also very important for measuring that the correct settings of ADC limits are used for touch detection.

23.2 Source Code Location

Table 23-1. Source code location

Driver	Location
TCHRES generic driver	`source/io/tchres`
TCHRES hardware-specific driver	`source/bsp/<board>/init_tchres.c`

23.3 Header Files

To use the TCHRES device driver, include the header file from *source/io/tchres* in your application or in the BSP file *bsp.h*. Use the header file according to the following table

Table 23-2. Header files

Driver	Header File
TCHRES driver	`tchres.h`

The file *tchres_prv.h* contains private constants and data structures that TCHRES device driver uses.

23.4 Installing Drivers

TCHRES device driver provides installation function *_io_tchres_install()* called by the user application. The driver installation routine calls *_io_dev_install_ext()* internally.

Example of *_io_dev_install* function call:

```
_io_tchres_install("tchscr:", &_bsp_tchscr_resisitve_init, &install_params);
```

The *_bsp_tchscr_resisitve_init* is the initialization structure containing information for TCHRES driver. The install_params points to the installation parameters structure containing information about ADC devices to be used for measuring.

Initialization structure contains, among other values, also the TCHRES_ADC_LIMITS_STRUCT which is important for measuring.

```
/* Adc limits struct */
typedef struct tchres_adc_limits
{
    uint16_t FULL_SCALE;
    int16_t  X_TOUCH_MIN;
    int16_t  Y_TOUCH_MIN;
    int16_t  X_TOUCH_MAX;
    int16_t  Y_TOUCH_MAX;
} TCHRES_ADC_LIMITS_STRUCT, * TCHRES_ADC_LIMITS_STRUCT_PTR;
```

A full scale should reflect ADC resolution. For example, for 12-bit ADC it should be set to 0xFFF. Minimum and maximum for x and y coordinate is used for filtering samples measured on X+ and Y+ electrodes. Samples out of this range will be interpreted as no touch.

```
/* install parameters - adc devices used for measuring on X+ and Y+ electrodes */
typedef struct tchres_install_param_struct
{
    char *ADC_XPLUS_DEVICE; /* ADC device for X+ electrode */
    char *ADC_YPLUS_DEVICE; /* ADC device for Y+ electrode */
} TCHRES_INSTALL_PARAM_STRUCT, * TCHRES_INSTALL_PARAM_STRUCT_PTR;
```

Installation parameters are used to provide string identifiers of ADC devices used for measuring on X+ and Y+ electrodes. Installation parameters should be provided by the user application which is also responsible for their opening prior to calling the driver installation routine.

23.5 Driver Services

TCHRES driver provides following services:

Table 23-3. Driver services

API	Calls
_io_fopen()	_tchres_open()
_io_fclose()	_tchres_close()
_io_ioctl()	_tchres_ioctl()

23.5.1 Opening TCHRES Device

Prior to using the touch screen device, it must be installed and opened. Since there is no need for any further work, the second parameter should be set to null as show in an example below.

```
FILE_PTR tchscr_dev = fopen("tchscr:", NULL);
```

Since there is no read or write function defined for the device, the communication is handled only by I/O control commands.

23.6 I/O Control Commands

This section describes I/O control commands used in *_io_ioctl()* calls on TCHRES device which are defined in *tchres.h*.

Table 23-4. I/O control commands

Command	Description	Parameters
IO_IOCTL_TCHSCR_GET_POSITION_RAW	Command measures touch position which is returned back in raw ADC values. Return code is either one of error code prefixed by TCHRES_ERROR_ or TCHRES_OK when touch was detected. Return codes are defined in tchres.h.	*param_ptr* - pointer to TCHRES_POSITION_STRUCT used for passing back touch result which is valid only on TCHRES_OK.
IO_IOCTL_TCHSCR_GET_RAW_LIMITS	Command returns ADC limits used for touch detection through parameter passed directly to ioctl as an argument. Return code is MQX_OK or TCHRES_ERROR_INVALID_PARAMETER.	*param_ptr* - pointer to TCHRES_ADC_LIMITS_STRUCT

23.7 Data Types

The following sections describe the data types used with the TCHRES driver.

23.7.1 TCHRES_INIT_STRUCT

Synopsis:

```
typedef struct tchres_init_struct {
   TCHRES_PIN_CONFIG_STRUCT PIN_CONFIG;
   TCHRES_ADC_LIMITS_STRUCT ADC_LIMITS;
   LWGPIO_PIN_ID ADC_CHANNEL_X_SOURCE;
   ADT_TRIGGER_MASK ADC_CHANNEL_X_TRIGGER;
   LWGPIO_PIN_ID ADC_CHANNEL_Y_SOURCE;
   ADT_TRIGGER_MASK ADC_CHANNEL_Y_TRIGGER;
} TCHRES_INIT_STRUCT, * TCHRES_INIT_STRUCT_PTR;
```

Parameters:

PIN_CONFIG - Pins connected to touch screen electrodes.

ADC_LIMITS - Limits for ADC used for touch detection.

ADC_CHANNEL_X_SOURCE - ADC channel for X+ electrode.

ADC_CHANNEL_X_TRIGGER - Trigger mask for X+ ADC channel.

ADC_CHANNEL_Y_SOURCE - ADC channel for Y+ electrode.

ADC_CHANNEL_Y_TRIGGER - Trigger mask for Y+ ADC channel.

23.7.2 TCHRES_PIN_CONFIG_STRUCT

Synopsis:

```
typedef struct tchres_pin_config_struct {
   LWGPIO_PIN_ID X_PLUS;
   LWGPIO_PIN_ID X_MINUS;
   LWGPIO_PIN_ID Y_PLUS;
   LWGPIO_PIN_ID Y_MINUS;
   TCHRES_PIN_FUNCT_STRUCT PIN_FUNCT;
} TCHRES_PIN_CONFIG_STRUCT, * TCHRES_PIN_CONFIG_STRUCT_PTR;
```

Parameters:

X_PLUS - X+ electrode GPIO pin definition.

X_MINUS - X- electrode GPIO pin definition.

Y_PLUS - Y+ electrode GPIO pin definition.

Y_MINUS - Y- electrode GPIO pin definition.

PIN_FUNCT - GPIO and ADC pin multiplexer masks.

23.7.3 TCHRES_PIN_FUNCT_STRUCT

Synopsis:

```
typedef struct tchres_pin_funct_struct {
   uint32_t X_PLUS_GPIO_FUNCTION;
   uint32_t X_PLUS_ADC_FUNCTION;
   uint32_t Y_PLUS_GPIO_FUNCTION;
   uint32_t Y_PLUS_ADC_FUNCTION;
   uint32_t X_MINUS_GPIO_FUNCTION;
   uint32_t Y_MINUS_GPIO_FUNCTION;
} TCHRES_PIN_FUNCT_STRUCT, * TCHRES_PIN_FUNCT_STRUCT_PTR;
```

Parameters:

X_PLUS_GPIO_FUNCTION - X+ electrode GPIO pin mux mask.

X_PLUS_ADC_FUNCTION - X+ electrode ADC pin mux mask.

Y_PLUS_GPIO_FUNCTION - Y+ electrode GPIO pin mux mask.

Y_PLUS_ADC_FUNCTION - Y+ electrode ADC pin mux mask.

X_MINUS_GPIO_FUNCTION - X- electrode GPIO pin mux mask.

Y_MINUS_GPIO_FUNCTION - Y- electrode GPIO pin mux mask.

23.7.4 TCHRES_ADC_LIMITS_STRUCT

Synopsis:

```
typedef struct tchres_adc_limits {
   uint16_t FULL_SCALE;
   int16_t  X_TOUCH_MIN;
   int16_t  Y_TOUCH_MIN;
   int16_t  X_TOUCH_MAX;
   int16_t  Y_TOUCH_MAX;
} TCHRES_ADC_LIMITS_STRUCT, * TCHRES_ADC_LIMITS_STRUCT_PTR;
```

Parameters:

FULL_SCALE - ADC resolution dependent parameter.

X_TOUCH_MIN - Min value for x-coordinate touch detection.

Y_TOUCH_MIN - Min value for y-coordinate touch detection.

X_TOUCH_MAX - Max value for x-coordinate touch detection.

Y_TOUCH_MAX - Max value for y-coordinate touch detection.

23.7.5 TCHRES_POSITION_STRUCT

Synopsis:

```
typedef struct tchres_position {
   int16_t X;
   int16_t Y;
} TCHRES_POSITION_STRUCT, * TCHRES_POSITION_STRUCT_PTR;
```

Parameters:

X - Touch position x-coordinate.

Y - Touch position y-coordinate.

23.7.6 Example

For basic use, see the MQX RTOS examples. The touch screen example is located in the directory mqx/examples/tchres. TCHRES demo application is written for tower system with connected TWR-LCD board.

TCHRES device typical usage is as follows:

- TCHRES device installation requires ADC device(s) to be opened:

```
adc_file = fopen(BSP_TCHRES_ADC_DEVICE, (const char*)&adc_init);
```

- Preparing install parameters (one ADC device for both X+ and Y+):

```
install_params.ADC_XPLUS_DEVICE = install_params.ADC_YPLUS_DEVICE =
BSP_TCHRES_ADC_DEVICE;
```

- When ADC device is successfully opened TCHRES can be installed:

```
_io_tchres_install("tchscr:", &_bsp_tchscr_resisitve_init, &install_params);
```

- Before reading from TCHRES device it has to be opened:

```
tchscr_dev = fopen("tchscr:", NULL);
```

- Read touch position using IOCTL:

```
if (_io_ioctl(tchscr_dev, IO_IOCTL_TCHSCR_GET_POSITION_RAW, &position) == TCHRES_OK) {
    printf("Touch detected (%d, %d)\n", position.X, position.Y);
}
```

23.7.7 Error Codes

Table 23-5. Error codes

Error code	Description
TCHRES_ERROR_INVALID_PARAMETER	Given parameter is invalid or NULL.
TCHRES_ERROR_NO_TOUCH	No touch detected, measured value is out of ADC limits range.
TCHRES_ERROR_TIMEOUT	When waiting for the screen surface preparation reach timeout.

Chapter 24
LWADC Driver

24.1 Overview

This section describes the Light-Weight ADC (LWADC) driver that accompanies MQX RTOS. This driver is a common interface for ADC modules.

LWADC driver implements custom API and does not follow the standard driver interface (I/O Subsystem). Therefore, it can be used before the I/O subsystem of MQX RTOS is initialized.

24.2 Source Code Location

The source files for the LWADC driver are located in `/mqx/source/io/lwadc` directory. *_lwadc* file prefix is used for all LWADC driver related files.

24.3 Header Files

To use LWADC driver, include the *lwadc.h* header file and the platform specific header file (e.g., *lwadc_k64.h*) in your application or in the BSP header file (*bsp.h*). The platform specific header should be included before *lwadc.h*.

24.4 API Function Reference

This section contains the function reference for the LWADC driver.

24.4.1 _lwadc_init()

Synopsis

```
uint32_t _lwadc_init
(
  const LWADC_INIT_STRUCT * init_ptr
)
```

Return Value

- TRUE (Success)
- FALSE (Failure)

Parameters

init_ptr [in] — Pointer to the device specific initialization information such as ADC device number, frequency, etc.

Description

This function initializes the ADC module according to the parameters given in the platform specific initialization structure. Call to this function does not start any ADC conversion. This function is normally called in the BSP initialization code. The initialization structures for particular devices are described in a separate subsection below.

24.4.2 _lwadc_init_input()

Synopsis

```
bool lwadc_init_input(
  LWADC_STRUCT_PTR lwadc_ptr,
  uint32_t         input
)
```

Parameters

lwadc_ptr [out] — Pointer to the application allocated context structure identifying the input.

input [in] — Input specification containing ADC device and MUX input.

Return Value

- TRUE (Success)
- FALSE (Failure)

Description

This function initializes the application allocated LWADC_STRUCT (which is device-specific) with all data needed later for quick control of particular input. The structure initialized here is used in all subsequent calls to other LWADC driver functions and uniquely identifies the input. To identify ADC input, platform specific input ID number is used. The function sets the ADC input to continuous conversion mode if not already in this mode.

24.4.3 _lwadc_read_raw()

Synopsis

```
bool _lwadc_read_raw
(
  LWADC_STRUCT_PTR lwadc_ptr,
  LWADC_VALUE *    outValue
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.

outValue [out] — Pointer to location to store read result.

Return Value

- TRUE (Success)
- FALSE (Failure)

Description

Read the current value of the ADC input and return the result without applying any scaling.

24.4.4 _lwadc_read()

Synopsis

```
bool _lwadc_read
(
  LWADC_STRUCT_PTR lwadc_ptr,
  LWADC_VALUE    * outValue
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.

outValue [out] — Pointer to a location to store the read result.

Return Value

- TRUE (success)
- FALSE (failure)

Description

Reads the current value of the ADC input, applies scaling according to preset parameters, see _lwadc_set_attribute() function below, and returns the result.

24.4.5 _lwadc_read_average()

Synopsis

```
bool _lwadc_read_average
(
  LWADC_STRUCT_PTR lwadc_ptr,
  uint32_t         num_samples,
  LWADC_VALUE    * outValue
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.

num_samples [in] — Number of samples to read.

outValue [out] — Pointer to location to store read result.

Return Value

- TRUE (success)
- FALSE (failure)

Description

Reads num_sample samples from the specified input and returns the scaled average reading.

24.4.6 _lwadc_set_attribute()

Synopsis

```
bool lwadc_set_attribute
(
```

```
  LWADC_STRUCT_PTR lwadc_ptr,
  LWADC_ATTRIBUTE  attribute,
  uint32_t         value
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.

attribute_id [in] — Attribute to enable/disable on the specified input.

value [out] — Value for the attribute.

Return Value

- TRUE (Success)
- FALSE (Failure)

Description

This function sets attributes for the specified ADC input. Attributes could include single/differential mode, reference, scaling numerator or denominator, etc. The following table summarizes all attributes.

Table 24-1. ADC attributes

ATTRIBUTE	Used to set or obtain:
LWADC_RESOLUTION	ADC Device resolution in steps.
LWADC_REFERENCE	ADC Reference voltage in millivolts.
LWADC_ FREQUENCY	ADC module base frequency, actual relation between this parameter and sampling rate parameter is device specific.
LWADC_DIVIDER	The input divider.
LWADC_DIFFERENTIAL	Enables channel as a differential input.
LWADC_POWER_DOWN	Power up or down the ADC Device.
LWADC_NUMERATOR	Numerator to be used on this channel for channel scaling.
LWADC_DENOMINATOR	Denominator to be used on this channel for channel scaling.
LWADC_FORMAT	Channel data format (such as left/right aligned).
LWADC_INPUT_CONVERSION_ENABLE	Enable or disable conversion for the input.

Note

Not all ADC devices support all attributes, nor all ADCs support a per-input setting of the attributes. Setting an attribute on one input may affect other or all inputs on a device.

24.4.7 _lwadc_get_attribute()

Synopsis

```
bool _lwadc_get_attribute
(
  LWADC_STRUCT_PTR lwadc_ptr,
  LWADC_ATTRIBUTE  attribute,
  uint32_t        *value
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.

attribute_id [in] — Attribute to obtain on the specified input.

value [out] — Pointer to the value for the attribute.

Return Value

- TRUE (Success)
- FALSE (Failure)

Description

This function gets attributes for the specified ADC input or for the ADC module as a whole. Attributes could include single/differential mode, reference, scaling numerator or denominator, etc. See also _lwadc_set_attribute().

24.4.8 _lwadc_wait_next()

Synopsis

```
bool lwadc_wait_next
(
  LWADC_STRUCT_PTR lwadc_ptr
)
```

Parameters

lwadc_ptr [in] — Context structure identifying the input.

Return Value

- TRUE (success)
- FALSE (failure)

Description

Waits for a new value to be available on the specified ADC input.

24.5 Data Types Used by the LWADC API

The following data types are used within the LWADC driver.

24.5.1 LWADC_INIT_STRUCT

This device-specific structure contains necessary parameters for initialization of ADC module on a particular platform.

Synopsis for Kinetis family:

```
typedef struct lwadc_init_struct {
uint8_t ADC_NUMBER;
    LWADC_CLOCK_SOURCE CLOCK_SOURCE;
    LWADC_CLOCK_DIV CLOCK_DIV;
    LWADC_HSC SPEED;
    LWADC_LPC POWER;
    uint8_t *CALIBRATION_DATA_PTR;
    uint32_t ADC_VECTOR;
    uint32_t ADC_PRIORITY;
    uint32_t REFERENCE;
}LWADC_INIT_STRUCT, * LWADC_INIT_STRUCT_PTR;
```

Parameters

ADC_NUMBER - The number of ADC peripheral, use adc_t enum from PSP.

CLOCK_SOURCE - The clock source.

CLOCK_DIV - The clock divisor for ADC.

SPEED - ADC high speed control, see ADC_HSC enum.

POWER - ADC low power control, see ADC_LPC enum.

CALIBRATION_DATA_PTR - The calibration data pointer.

ADC_VECTOR - ADC interrupt vector.

ADC_PRIORITY - ADC interrupt vector.

REFERENCE - Preset reference voltage in millivolts, see LWADC_REFERENCE attribute.

24.5.2 LWADC_STRUCT

Device specific context structure keeping data for fast access to the device. A pointer to this structure is used to refer to a particular ADC input in LWADC API calls.

24.5.3 Other Data Types

```
typedef enum {
    LWADC_RESOLUTION=1,
    LWADC_FREQUENCY,
    LWADC_DIVIDER,
    LWADC_DIFFERENTIAL,
    LWADC_POWER_DOWN,
    LWADC_NUMERATOR,
    LWADC_DENOMINATOR,
    LWADC_FORMAT
}  LWADC_ATTRIBUTE;
```

Members of this enum are used to refer to LWADC attributes in calls to _lwadc_set_attribute() and _lwadc_get_attribute().

The format identifiers for LWADC_FORMAT attribute are defined as macros:

LWADC_FORMAT_LEFT_JUSTIFIED

LWADC_FORMAT_RIGHT_JUSTIFIED

24.6 Example

An example application demonstrating LWADC usage is provided. The example application can be found in `/mqx/examples/lwadc` directory.

Chapter 25
HMI

25.1 Overview

This section describes the HMI, Human Machine Interface, driver which is part of the MQX RTOS driver set.

The HMI driver provides an API for configuration and control of input controls such as buttons or touch electrodes and output controls such as LEDs. The HMI driver consists of two abstract layers, as shown in the figure below and described in detail in this section. The abstract and layered API can be used by various kinds of drivers implementing the human-to-machine interface.

In the current MQX RTOS version, there is one instance of the HMI driver called "BTNLED" which can be used by MQX RTOS applications to transparently handle the following input and output controls:

- Physical push buttons connected to GPIO pins and accessed with the LWGPIO driver.
- Touch electrodes handled by the Freescale Touch Sensing Library. See www.freescale.com/tss.
- LEDs connected to GPIO pins accessed with the LWGPIO driver.

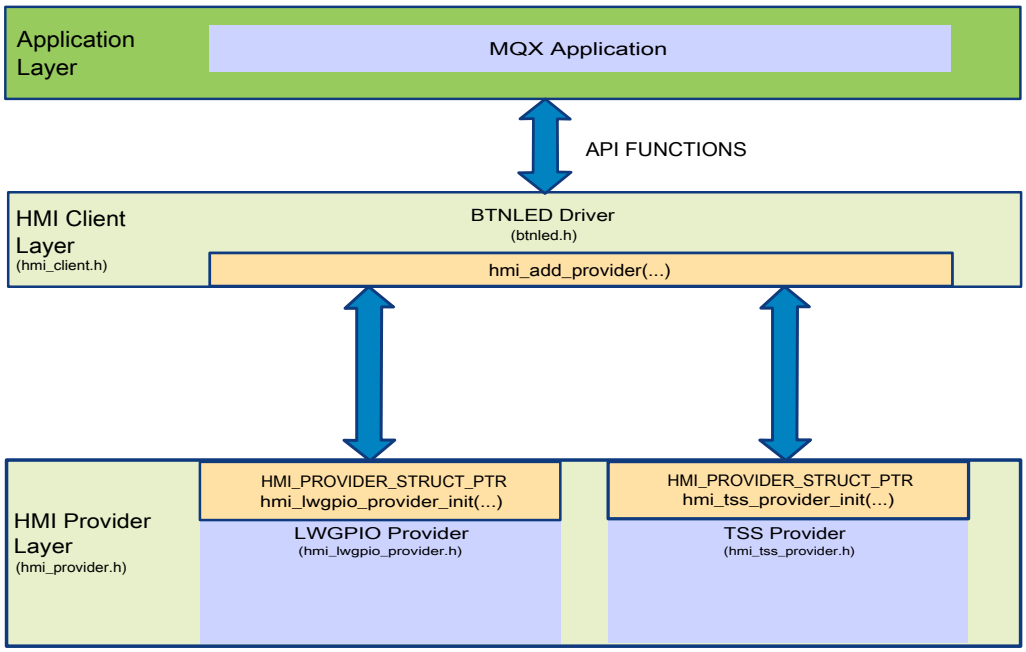

Figure 25-1. HMI Layers

HMI driver layers are designed to handle more instances of physical human-machine interfaces (so called HMI providers). For example the BTNLED driver could be extended to use PS-2 or USB Keyboards for key handling and/or external USB LED panels for signaling. The HMI API design enables such new interfaces to be implemented and used by an application without any changes in the application code.

With the HMI layered approach, there can also be other implementations of the HMI driver with a different behavior but still reusing the existing interface providers. An example of such an HMI driver instance, which is not yet implemented in the current MQX RTOS version, is a Keyboard driver which provides services of a common PC-like keyboard on top of existing providers for physical GPIO push buttons, touch electrodes, or USB keyboards.

The following sections describe the general HMI layers as well as the BTNLED driver API which can be used in the MQX RTOS applications. Use of the BTNLED driver is also demonstrated in the HMI example application located in mqx/examples/hmi folder.

The BTNLED driver and the touch sensing HMI provider driver is available for all BSPs and platforms supported by the Freescale Touch Sensing Library version available at the time of thisd release. It can be easily extended to other platforms supported by the subsequent versions of the TSS Library.

25.2 HMI Driver Layers

This section describes the HMI driver layers.

25.2.1 HMI Client Layer

HMI Client Layer is used as an interface between the HMI driver instance and the MQX user application. The layer provides a set of functions to manage and communicate with one or more HMI interface providers in a transparent way.

The BTNLED driver, described in more detail in the following sections, is a good example of an HMI Client. The BTNLED driver is part of the driver set in the BSP project and enables the MQX applications to access push buttons, touch electrodes, and LEDs in a transparent way independent on physical board design.

25.2.2 HMI Provider Layer

The HMI Provider Layer enables the hardware abstraction for the HMI Client Layer. This layer API enables any Client to attach to a provider and access its HMI controls in a polled or interrupt-driven way.

Two providers are implemented in the current version of MQX RTOS as follows:

- **HMI LWGPIO Provider** implements access to buttons and LEDs connected to microcontroller General-purpose I/O pins. The provider uses the MQX RTOS LWGPIO driver internally to control the GPIO ports and pins. This provider can handle both active-low and active-high push buttons and LEDs. It is typically a BSP code which initializes the provider controls according to a physical board connection.
- **TSS Provider** wraps the Freescale Touch Sensing library and enables a button-like handling of electrodes based on capacitance change detection. The TSS Library is a separate software package included in a binary form within the MQX RTOS distribution. Check the www.freescale.com/tss for the latest version of the library, detailed documentation, and more examples of touch sensing implementation.

25.2.3 HMI UID

Each input or output control handled by the HMI Driver is identified by a 32-bit identifier called 'UID'. There is a common naming convention for general HID elements in the driver:

- **HMI_BUTTON_n** used for buttons, regardless if they are physical push buttons or touch electrodes. All MQX BSPs which implement the HMI driver assign the HMI_BUTTON_n constants sequentially:
  - HMI_BUTTON_1 assigned to the first on-board touch electrode
  - followed by other on-board touch electrodes
  - followed by physical on-board push buttons

  Some boards also enable so-called TWRPI electrode daughter cards to be attached. If this is the case, the BSP provides an API to re-map the electrodes of the selected TWRPI module to HMI_BUTTON_1..n and use them as alternatives to on-board buttons.
- **HMI_SLIDER_n** used for volume-up/down-like controls either implemented as electrical potentiometers or virtual ones based on touch position evaluation.
- **HMI_ROTARY_n** a "jog-dial" similar to the Slider control without explicit minimum and maximum position.
- **HMI_LED_n** used for LEDs or other visual on/off signals provided on board.

Internally, the UID constants are encoded as a combination of so-called usage table identifier and a usage ID, both 16-bit values.

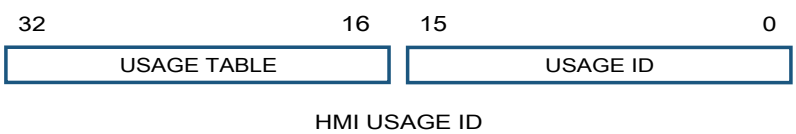

Figure 25-2. HMI Usage ID

25.3 Source Code Location

HMI driver is a part of the IO driver set and is compiled along with the other drivers into the BSP library. HMI API functions, source code, and data types are defined in source files located in the *source/io/hmi* directory.

Each BSP which makes use of the HMI driver, the BTNLED driver, implements also an *init_hmi.c* file in the BSP directory. This file contains board-specific initialization of LWGPIO and TSS providers and support for TWRPI touch-sensing electrode modules.

25.4 Header Files

To use the HMI functionality of the BTNLED driver, include the *bsp.h* header file into your application. The file contains all include statements needed for the HMI functionality.

25.5 API Function Reference

This section provides a function reference for the HMI and BTNLED drivers.

25.5.1 HMI CLIENT

This section describes the HMI clients.

25.5.1.1 hmi_client_init()

Synopsis

```
HMI_CLIENT_STRUCT_PTR hmi_client_init(void)
```

Parameters

None.

Return Value

Pointer to HMI_CLIENT_STRUCT structure used as a handle to the HMI Client instance.

Description

This function allocates and initializes memory for an instance of the HMI Client. This function is typically used internally in an initialization function of real HMI Client driver implementation - for example the BTNLED driver.

25.5.1.2 hmi_add_provider()

Synopsis

```
bool hmi_add_provider
(
 HMI_CLIENT_STRUCT_PTR   client_handle,
 HMI_PROVIDER_STRUCT_PTR provider_handle
)
```

Parameters

client_handle [in] — Client structure handle.

provider_handle [in] — Provider structure handle.

Return Value

- TRUE - The HMI Provider has been successfully registered with the Client.
- FALSE - Failed to register the provider instance. Typically this is an out of memory issue.

Description

This function registers an HMI Provider instance in the Client's provider list. This function is typically used in the BSP initialization code to bind HMI providers to the HMI Client driver (e.g., to bind LWGPIO or TSS provider to the BTNLED driver).

25.5.1.3 hmi_remove_provider()

Synopsis

```
bool hmi_remove_provider
(
 HMI_CLIENT_STRUCT_PTR   client_handle,
 HMI_PROVIDER_STRUCT_PTR provider_handle
)
```

Parameters

client_handle [in] — Client structure handle.

provider_handle [in] — Provider structure handle.

Return Value

- TRUE - The HMI Provider has been successfully unregistered from the Client.
- FALSE - Failed to unregister the provider. Invalid Client or Provider handle specified.

Description

This function unregisters the HMI Provider from the Client's provider list.

25.5.2 BTNLED

This section describes the BTNLED HMI clients.

25.5.2.1 btnled_init()

Synopsis

```
HMI_CLIENT_STRUCT_PTR btnled_init(void)
```

Parameters

None.

Return Value

HMI_CLIENT_STRUCT_PTR handle to newly created BTNLED HMI Client.

Description

This function allocates memory for the BTNLED HMI client structure and initializes it. This function must be called first before using other driver API. The returned pointer serves as a handle to the BTNLED client instance and needs to be passed as a "handle" argument to other BTNLED driver functions.

25.5.2.2 btnled_deinit()

Synopsis

```
uint32_t btnled_deinit
(
 HMI_CLIENT_STRUCT_PTR handle
)
```

Parameters

handle [in] — Client structure handle.

Return Value

- MQX_OK

Description

This function de-allocates memory which used by the BTNLED HMI Client.

25.5.2.3 btnled_poll()

Synopsis

```
void btnled_poll
(
 HMI_CLIENT_STRUCT_PTR handle
)
```

Parameters

handle [in] — Client structure handle.

Return Value

None.

Description

This function polls all interface providers attached to the BTNLED Client. This gives each provider a chance to evaluate input signals and notify the client about any change. Each provider implements different poll functionality. For example the TSS provider invokes the TSS_Task routine and handles the TSS event callbacks.

25.5.2.4 btnled_get_value()

Synopsis

```
bool btnled_get_value
(
 HMI_CLIENT_STRUCT_PTR handle,
 uint32_t              uid,
 uint32_t             *value
)
```

Parameters

handle [in] — Client structure handle.

uid [in] — UID identifier of an input control.

value [out] — Returns the immediate state value for a specified.

Return Value

- TRUE - the value of the UID control has been successfully obtained.
- FALSE - the UID control is not available in any registered provider.

Description

This function gets a value of a specified UID control. For button-like controls, the returned value reflects the state of the button. It is a non-zero value when button is pressed, zero if button is released.

For a slider or rotary controls, the returned integer value reflects the current finger position within the control.

25.5.2.5 btnled_set_value()

Synopsis

```
bool  btnled_set_value
(
 HMI_CLIENT_STRUCT_PTR handle,
 uint32_t              uid,
 uint32_t              value
)
```

Parameters

handle [in] — Client structure handle.

uid [in] — UID identifier of an input control.

value [in] — State value to set.

Return Value

- TRUE - the value of the UID control has been successfully set.
- FALSE - the UID control is not available in any registered provider.

Description

This function sets the control value. It is currently supported for LED output control only. Use one of HMI_VALUE_ON or HMI_VALUE_OFF constants.

25.5.2.6 btnled_toogle()

Synopsis

```
bool btnled_toggle
(
 HMI_CLIENT_STRUCT_PTR handle,
 uint32_t              uid
)
```

Parameters

handle [in] — Client structure handle.

uid [in] — UID identifier of an input control.

Return Value

- TRUE - the value of the UID control has been successfully set.
- FALSE - the UID control is not available in any registered provider.

Description

This function toggles the control value. It is currently supported for LED output control only.

25.5.2.7 btnled_add_clb()

Synopsis

```
void *btnled_add_clb
(
 HMI_CLIENT_STRUCT_PTR handle,
 uint32_t              uid,
 uint32_t              state,
 void (_CODE_PTR_      function)(void*),
 void                 *callback_parameter
)
```

Parameters

handle [in] — Client structure handle.

uid [in] — UID identifier of an input control.

state [in] — State which causes the callback to be invoked.

function [in] — Callback function to be invoked when the state changes.

callback_parameter [in] — Parameter to be passed into the callback function.

Return Value

Handle to registered callback instance. Use this value to un-register the callback function.

Description

This function registers a callback function to handle state changes for a given UID control. If the control state changes and gets equal to the selected state, the registered callback function is invoked. Use the HMI_VALUE_PUSH or HMI_VALUE_RELEASE states for button-like controls. Use HMI_VALUE_MOVEMENT state for slider and rotary controls.

25.5.2.8 btnled_remove_clb()

Synopsis

```
bool btnled_remove_clb
(
 HMI_CLIENT_STRUCT_PTR handle,
 void                  *comp_clbreg
)
```

Parameters

handle [in] — Client structure handle.

comp_clbred [in] — Handle to registered callback instance.

state [in] — State which causes the callback to be invoked.

function [in] — Callback function to be invoked when the state changes.

callback_parameter [in] — Parameter to be passed into the callback function.

Return Value

- TRUE - The callback function has been successfully un-registered.
- FALSE - Could not un-register the callback function, invalid callback handle specified.

Description

This function un-registers the callback function previously registered to handle control state changes.

25.5.3 DATA TYPES

This section describes the HMI data types.

25.5.3.1 HMI_PROVIDER_STRUCT

A pointer to this structure represents an instance of the HMI Provider. Each provider instance is allocated and initialized by the provider-specific initialization function. The user application typically does not use this type. It is used internally by the HMI Client layer.

25.5.3.2 HMI_CLIENT_STRUCT

A pointer to this structure represents an instance of the HMI Client. Each client instance is allocated and initialized by the client-specific initialization function. The user application uses the pointer as a handle to selected HMI Client instance and passes it to all Client API functions.

25.5.3.3 HMI_TSS_INIT_STRUCT

HMI_TSS_INIT_STRUCT is the HMI TSS Provider initialization structure. An array of these structures is used in the BSP code to create and initialize any TSS provider instance. Such an initialization array should be always terminated with a zeroed structure.

The structure contains the UID identifier (UID) to be assigned to a touch sensing electrode or TSS control. The FLAG member is reserved for future use and should be zeroed.

```
typedef struct hmi_tss_init_struct
{
    uint32_t  UID;
    uint8_t   FLAG;
} HMI_TSS_INIT_STRUCT, * HMI_TSS_INIT_STRUCT_PTR;
```

- For TSS Keypad controls which consist of multiple electrodes, the order of the structures in the array should match the order of electrodes configured in the TSS configuration. A HMI Provider instance for TSS Keypad control is initialized with *hmi_tss_keypad_provider_init()* function.
- For TSS Rotary and Slider controls, the initialization array typically consists of a single element assigning the UID to the whole TSS control. A TSS Provider instance for TSS Rotary control is initialized with *hmi_tss_rotary_provider_init()* function, a provider for TSS Slider control is initialized with *hmi_tss_slider_provider_init()* function.

25.5.3.4 HMI_TSS_SYSTEM_CONTROL_STRUCT

This structure contains initial settings for the TSS Library used internally by the HMI TSS providers. The TSS Library should be initialized first by calling the *hmi_tss_init()* function before any TSS HMI provider can be created.

See *TSS Library User Guide* for more details on configuration and electrode sensitivity parameters used in this structure.

```
typedef struct hmi_tss_system_control_struct
{
    uint8_t        SYSTEM_CONFIG;
    uint8_t        SYSTEM_TRIGGER;
    uint8_t        NUMBER_OF_SAMPLES;
    const uint8_t *SENSITIVITY_VALUES;
} HMI_TSS_SYSTEM_CONTROL_STRUCT, * HMI_TSS_SYSTEM_CONTROL_STRUCT_PTR;
```

25.5.3.5 HMI_LWGPIO_INIT_STRUCT

The HMI_LWGPIO_INIT_STRUCT is the HMI LWGPIO Provider initialization structure. An array of these structures is used in the BSP code to create and initialize any instance of LWGPIO provider. The array should be always terminated with a zeroed structure.

```
typedef struct hmi_lwgpio_init_struct
{
    uint32_t UID;
    uint32_t PID;
    uint32_t FUNC;
    uint32_t FLAG;
}HMI_LWGPIO_INIT_STRUCT, * HMI_LWGPIO_INIT_STRUCT_PTR;
```

For each button or LED, the structure binds the UID identifier with the GPIO port and pin defined by port and pin ID (PID) and LWGPIO multiplexer function setting (FUNC). The FLAG parameter can be used to define active-low or active-high pins as well as to enable internal pull-up or pull-down resistors for the button state sensing.

See the *io/hmi/hmi_lwgpio_provider.h* file for more details on the supported FLAG values.

25.6 Example

The example of the HMI that demonstrates how to use HMI API functions is provided along with the MQX installation and it is located in *mqx/examples/hmi* directory.

The default settings of BTNLED client can be found in the corresponding BSP directory in the source file *init_hmi.c*.

Chapter 26
Debug I/O Driver

26.1 Overview

The debug I/O driver implements a data communication channel between the client and host during a debugger session. Not all debugger tools support this feature and not all support bidirectional communication. Typically the tools support output communication only (target writing to debugger console).

The driver is currently supporting the Semihost and ITM mode for Kinetis ARM® Cortex®-M4 processors only. When a processor reaches BKPT semihost instruction, the processor enters a halt state and waits until the debugger finishes its job. Tasks and interrupts cannot be performed during the halt state. File operations (open, write, read, ioctl, close) are protected with a semaphore. The driver can be safely accessed from multiple tasks and can be set as a default "stdout" and "stdin" channel.

26.2 Source Code Location

The source code for debug driver is located in *source/io/debug* directory.

26.3 Header Files

To use a debug driver in your application, include the *bsp.h* header file, which includes the the main driver header file *debug.h*.

The file *debug_prv.h* file contains private constants and data structures which the driver uses. You should typically include this file if you change or enhance the driver itself. You may also want to look at the file as you debug your application.

26.4 Installing Drivers

The debug driver provides an installation function that either the BSP or the application calls. The function then calls **_io_dev_install_ext()** internally. Usually, the **_io_debug_install()** installation function is called from *init_bsp.c* if enabled by **BSPCFG_ENABLE_IODEBUG** configuration option in *user_config.h*

Example of the **_io_debug_install** function call:

```
#if BSPCFG_ENABLE_IODEBUG
    _io_debug_install("iodebug:", &_bsp_iodebug_init);
#endif
```

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

26.5 Initialization Record

When installing the debug driver, the pointer to initialization record is passed. The following code is an example as it can be found in *init_debug.c*:

```
const IODEBUG_INIT_STRUCT _bsp_iodebug_init = {
    IODEBUG_MODE_SEMIHOST,    /* Driver mode */
    127                       /* Length of buffered data */
    IODEBUG_NOFLUSH_CHAR      /* Default flush character */
};
```

26.6 IODEBUG_INIT_STRUCT

Synopsis

```
typedef struct {
    uint32_t MODE;
    uint32_t DATA_LENGTH;
    char     FLUSH_CHAR;
} IODEBUG_INIT_STRUCT, * IODEBUG_INIT_STRUCT_PTR;
```

Parameters

MODE - Selects the mode of operation. Available modes are:

- IODEBUG_MODE_SEMIHOST - Output is performed by "semihosting" mechanism. User messages are transferred as an exception by executing set of special instructions. During the data transfer, the processor has to be halted. Transferring each character independently can be really slow. To speed up the communication, you should use the buffer. Data is transferred together as a block.
- IODEBUG_MODE_ITM - Output is performed by using ITM, Instrumentation Trace Macrocell, one of CoreSight components. The ITM interface enables writing custom messages as trace information.

DATA_LENGTH - The buffer length. The buffer is enabled if DATA_LENGTH structure member is non zero.

FLUSH_CHAR - Flush character, for instance '\n'.

26.7 Driver Services

The debug driver provides these services:

Table 26-1. Debug driver services

API	Calls
_io_fopen()	_io_debug_open()
_io_fclose()	_io_debug_close()
_io_read()	_io_debug_read()
_io_write()	_io_debug_write()
_io_ioctl()	_io_debug_ioctl()

26.8 Using IOCTL Commands

This section describes the I/O control commands that are used when calling **_io_ioctl**() for the IO debug driver.

26.8.1 General IOCTL commands

Table 26-2. General IOCTL commands

Command	Description	Params
IO_IOCTL_FLUSH_OUTPUT	Immediately flush ouput buffer.	NULL

26.8.2 Driver specific IOCTL commands

Table 26-3. Driver specific IOCTL commands

Command	Description	Params
IO_IOCTL_IODEBUG_SET_FLUSH_CHAR	Set flush character.	Pointer to flush character. For example: "\n"

26.9 Example

This example shows opening the debug I/O port and setting the handle as its standard output channel.

```
FILE_PTR fh_ptr;
if(NULL == (fh_ptr = fopen("iodebug:", NULL))) {
  printf("Cannot open the debug output\n");
} else {
  _io_set_handle(IO_STDOUT, fh_ptr);
  printf("This is printed to the debug output\n");
}
fflush(stdout);
if (fh_ptr != NULL) {
  fclose(fh_ptr);
}
```

Chapter 27
I2S Driver

27.1 Overview

This section describes the I2S device driver.

The driver uses interrupts of SSI bus for ColdFire family, or I2S peripheral module for Kinetis family and its hardware FIFO buffers.

27.2 Source Code Location

The source code for the I2S driver is located in *source/io/i2s* directory.

27.3 Header Files

To use I2S driver, include the *i2s.h, is2_audio.h* header files and the platform specific header file, *i2s_mcf54xx.h*, into your application or into the BSP header file, *bsp.h*.

27.4 Installing Drivers

The I2S driver provides installation functions that either the BSP or the application calls. **_ki2s_int_install()** function is used for Kinetis devices with the I2S peripheral module and **_mcf54xx_i2s_int_install()** function serves for mcf54xx devices. One of these functions is typically called from *init_bsp.c* if enabled by **BSPCFG_ENABLE_II2Sx** configuration option in *user_config.h*

Example of the **_mcf54xx_i2s_int_install** function call:

```
#if BSPCFG_ENABLE_II2S0
    _mcf54xx_i2s_int_install("ii2s0:", &_bsp_i2s0_init);
#endif
```

This code can be found typically in */mqx/source/bsp/<platform>/init_bsp.c* file.

After installation, the driver gets full control of SSI periphery and switches it to I2S compatible mode.

27.5 Initialization Record

When installing the I2S driver, the pointer to initialization record is passed. The following code is an example, as it can be found in *init_i2s0.c:*

```
const AUDIO_DATA_FORMAT _bsp_audio_data_init = {
   AUDIO_BIG_ENDIAN,       /* Endian of input data */
   AUDIO_ALIGNMENT_LEFT,   /* Aligment of input data */
   8,                      /* Bit size of input data */
   1,                      /* Sample size in bytes */
      1                       /* Number of channels */
};
const MCF54XX_I2S_INIT_STRUCT _bsp_i2s0_init = {
   0,                       /* I2S channel    */
   BSP_I2S0_MODE,           /* I2S mode       */
   BSP_I2S0_DATA_BITS,      /* Number of valid data bits*/
   BSP_I2S0_CLOCK_SOURCE,   /* SSI_CLOCK source*/
   FALSE,                   /* Data is stereo */
   FALSE,                   /* Transmit dummy data */
   BSP_I2S0_INT_LEVEL,      /* Interrupt level to use */
   BSP_I2S0_BUFFER_SIZE,    /* Tx buffer size */
   &_bsp_audio_data_init    /* I/O data format */
};
```

For detailed description of used data types, see Data Types Used by the I2S Driver.

27.6 Driver Services

The I2S device driver provides the following services:

Table 27-1. I2S device driver services

API	Calls	Description
_io_fopen()	_io_i2s_open()	Initializes hardware and calls GPIO init code to initialize peripheral and its information structure and buffers.
_io_fclose()	_io_i2s_close()	Deinitialize and close device driver. Frees all buffers and structures used by driver.
_io_read()	_io_i2s_read()	Used to transmit data over I2S bus.
_io_write()	_io_i2s_write()	Used to receive data from I2S bus.
_io_ioctl()	_io_i2s_ioctl()	Used to set/get peripheral configuration.

27.7 Using I/O Control Commands

This section describes the I/O control commands that are used when calling **_io_ioctl()** for the I2S driver.

Each of the listed functions has implemented input parameter check. If the IOCTL command requires a parameter but gets empty pointer with a NULL value, then I2S_ERROR_INVALID_PARAMETED error code is returned. When the given parameter has an incorrect range, the I2S_ERROR_PARAM_OUT_OF_RANGE error code is returned.

Table 27-2. I/O control commands

Command	Description	Params
IO_IOCTL_I2S_SET_MODE_MASTER	Sets driver to master mode. During switch process, processor's particular pin is configured as a master clock signal receiver or transmitter depending on whether the internal or the external source of this signal is selected.	none (NULL)
IO_IOCTL_I2S_SET_MODE_SLAVE	Sets driver to a slave mode. In this mode, device registers are set to receive clock signals such as frame sync, master clock, and bit clock from an external source.	none (NULL)
IO_IOCTL_I2S_SET_CLOCK_SOURCE_INT	Switches to internal master clock source. Clock signal is taken from the bus clock using configurable dividers. Then, according to divider settings, all derived signals setting is recomputed.	none (NULL)
IO_IOCTL_I2S_SET_CLOCK_SOURCE_EXT	Switches to external master clock source.	none (NULL)

Table continues on the next page...

Table 27-2. I/O control commands (continued)

Command	Description	Params
	I2S device will disconnect from the bus clcok and all bus signals must be connected externally.	
IO_IOCTL_I2S_SET_DATA_BITS	Sets data word length. Command parameter contains desired data length. Possible values are 8, 10, 12, 16, 18, 20, 22, or 24 bits. Data word length reduction is made by hardware so various settings have no impact on the driver's performance.	*uint8_t**
IO_IOCTL_I2S_DISABLE_DEVICE	Disables I2S device. First, interrupts are disabled. After that, the whole device is disabled.	none (NULL)
IO_IOCTL_I2S_ENABLE_DEVICE	Enables I2S device. First, interrupts are enabled. After that, the whole device is enabled.	none (NULL)
IO_IOCTL_I2S_SET_MCLK_FREQ	Sets master clock frequency. If internal clock source is selected, driver changes the master clock divider to match the requested frequency. If exact match is impossible, the closest possible master clock frequency is selected and set. This frequency is stored in the information structure from which the value can be read by the appropriate command. All other frequencies which are derived from the master clock signal are computed in a similar way.	uint32_t*

Table continues on the next page...

Table 27-2. I/O control commands (continued)

Command	Description	Params
	If the external clock source is selected, then its value is stored directly in the information structure as a parameter.	
IO_IOCTL_I2S_SET_FS_FREQ	Sets frame sync frequency. This signal is derived from the bit clock signal. The frequency is passed to the command as a parameter and internal dividers are used to set it.	uint32_t*
IO_IOCTL_I2S_TX_DUMMY_ON	Enables transmitting without input data. Transmitted data is generated by the driver and consists of 440Hz sine signal sampled with the synchronization signal frequency. The "dummy" data is stored in the driver internal memory.	none (NULL)
IO_IOCTL_I2S_TX_DUMMY_OFF	Disables transmitting without input data.	none (NULL)
IO_IOCTL_I2S_GET_MODE	Gets pointer to actual mode. Stores the mode information into a variable pointed at by the parameter. Stored value equals either I2S_MODE_SLAVE or I2S_MODE_MASTER.	uint8_t*
IO_IOCTL_I2S_GET_CLOCK_SOURCE	Gets pointer to the actual master clock source. Stores the master clock source information into a variable pointed at by the parameter. Stored value equals either I2S_CLK_INT, internal clock source, or I2S_CLK_EXT, external clock source.	*uint8_t**

Table continues on the next page...

Table 27-2. I/O control commands (continued)

Command	Description	Params
IO_IOCTL_I2S_GET_DATA_BITS	Copies value of actual data word length to a variable designated by the parameter.	*uint8_t**
IO_IOCTL_I2S_GET_MCLK_FREQ	Copies actual master clock frequency in Hz to a variable designated by the parameter.	*uint32_t**
IO_IOCTL_I2S_GET_BCLK_FREQ	Copies actual bit clock frequency to a variable designated by the parameter. This frequency is derived from the synchronization signal frequency and the data word length. The returned frequency is in Hz.	*uint32_t**
IO_IOCTL_I2S_GET_TX_DUMMY	If a device is transmitting without input data, this command copies TRUE to a variable designated by the parameter.	*bool**
IO_IOCTL_I2S_GET_FS_FREQ	Copies the value of the frame sync signal frequency in Hz to a variable designated by the parameter.	*uint32_t**
IO_IOCTL_I2S_GET_STATISTICS	Copies the actual driver I/O statistics to a variable designated by the parameter.	*I2S_STATISTICS_STRUCT_PTR*
IO_IOCTL_I2S_SET_TXFIFO_WATERMARK	Sets transmitter watermark value for both transmit FIFO buffers. When the number of samples in a buffer drops below the watermark value, an interrupt is generated. Note that setting this value can affect frequency of interrupts generated by the driver. Ensure that the watermark is not set too low. Otherwise, it could cause the buffer	*uint8_t**

Table continues on the next page...

Table 27-2. I/O control commands (continued)

Command	Description	Params
	underflow if it is not re-filled in time. The default watermark value is set to 5. The range is 0 - 15.	
IO_IOCTL_I2S_SET_RXFIFO_WATERMARK	Sets the receiver watermark value for the receiver. It works the same way as a command for the transceiver, with the exception that the interrupt is generated when the number of samples in the buffer rises above the watermark value. By default, the receiver watermark value is set to 8.	*uint8_t**
IO_IOCTL_I2S_GET_TXFIFO_WATERMARK	Gets the actual transmitter watermark value.	*uint8_t**
IO_IOCTL_I2S_GET_RXFIFO_WATERMARK	Gets the actual receiver watermark value.	*uint8_t**
IO_IOCTL_I2S_SET_CLK_ALWAYS_ENABLED_ON	Enables permanent transmission on non-data signals such as clock, synchronization and master clock signal.These signals may be used by devices connected to a bus as clock sources.	none (NULL)
IO_IOCTL_I2S_SET_CLK_ALWAYS_ENABLED_OFF	Disables permanent transmission of non-data signals.	none (NULL)
IO_IOCTL_I2S_GET_CLK_ALWAYS_ENABLED	Gets the actual permanent transmission of non-data signal settings. If transmitting is enabled, the command returns TRUE. Otherwise, it returns FALSE.	bool*
IO_IOCTL_I2S_CLEAR_STATISTICS	Clears statistics.	none (NULL)
IO_IOCTL_AUDIO_SET_IO_DATA_FORMAT	Sets input and output data format.	AUDIO_DATA_FORMAT_PTR

Table continues on the next page...

Table 27-2. I/O control commands (continued)

Command	Description	Params
IO_IOCTL_AUDIO_GET_IO_DATA_FORMAT	Gets input and output data format.	AUDIO_DATA_FORMAT_PTR

27.8 Data Types Used by the I2S Driver

This section describes the data types that are used within the I2S driver.

27.8.1 MCF54XX_I2S_INIT_STRUCT, KI2S_INIT_STRUCT

Synopsis

```
typedef struct mcf54xx_i2s_init_struct
{
    uint8_t                   CHANNEL;
    uint8_t                   MODE;
    uint8_t                   DATA_BITS;
    uint8_t                   CLOCK_SOURCE;
    bool                      STEREO;
    bool                      TX_DUMMY;
    _int_level                LEVEL;
    uint32_t                  BUFFER_SIZE;
    AUDIO_DATA_FORMAT  const *IO_FORMAT;
} MCF54XX_I2S_INIT_STRUCT, * MCF54XX_I2S_INIT_STRUCT_PTR;
typedef struct ki2s_init_struct
{
   uint8_t                  CHANNEL;
   uint8_t                  MODE;
   uint8_t                  DATA_BITS;
   uint8_t                  CLOCK_SOURCE;
   bool                     STEREO;
   bool                     TX_DUMMY;
   _int_level               LEVEL;
   uint32_t                 BUFFER_SIZE;
   AUDIO_DATA_FORMAT const *IO_FORMAT;
} KI2S_INIT_STRUCT, * KI2S_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - I2S hardware channel.

MODE - Driver mode (master / slave).

DATA_BITS - Number of valid data bits.

CLOCK_SOURCE - Master clock source number.

TX_DUMMY - Transmit without input data (yes /no).

LEVEL - Driver's interrupt level.

BUFFER_SIZE - Driver's internal buffer size.

AUDIO_DATA_FORMAT - Pointer to input / output data format structure.

27.8.2 I2S_STATISTICS_STRUCT

Synopsis

```
typedef struct i2s_statistics_struct
{
    uint32_t INTERRUPTS;
    uint32_t UNDERRUNS_L;
    uint32_t UNDERRUNS_R;
    uint32_t OVERRUNS_L;
    uint32_t OVERRUNS_R;
    uint32_t RX_PACKETS;
    uint32_t TX_PACKETS;
    uint32_t PACKETS_PROCESSED_L;
    uint32_t PACKETS_QUEUED_L;
    uint32_t PACKETS_REQUESTED_L;
    uint32_t PACKETS_PROCESSED_R;
    uint32_t PACKETS_QUEUED_R;
    uint32_t PACKETS_REQUESTED_R;
} I2S_STATISTICS_STRUCT, * I2S_STATISTICS_STRUCT_PTR;
```

Parameters

INTERRUPTS - Number of driver generated interrupts.

UNDERRUNS_L - Number of left buffer underflow.

UNDERRUNS_R - Number of right buffer underflow.

OVERRUNS_L - Number of left buffer overflow.

OVERRUNS_R - Number of right buffer overflow.

RX_PACKETS - Number of samples received.

TX_PACKETS - Number of samples transmitted.

PACKETS_PROCESSED_L - Number of sent left channel samples.

PACKETS_QUEUED_L - Number of buffered left channel samples.

PACKETS_REQUESTED_L - Requested number of sent left channel samples.

PACKETS_PROCESSED_R - Number of sent right channel samples.

PACKETS_QUEUED_R - Number of buffered right channel samples.

PACKETS_REQUESTED_R - Requested number of sent right channel samples.

The content of the I2S_STATISTICS_STRUCT structure is cleared automatically when the driver is closed with _io_fclose(). It can also be cleared manually with IO_IOCTL_I2S_CLEAR_STATISTICS command.

27.8.3 AUDIO_DATA_FORMAT

Synopsis

```
typedef struct audio_data_format
{
    uint8_t ENDIAN;
    uint8_t ALIGNMENT;
    uint8_t BITS;
    uint8_t SIZE;
    uint8_t CHANNELS;
} AUDIO_DATA_FORMAT, * AUDIO_DATA_FORMAT_PTR;
```

Parameters

ENDIAN - Data endianness, either AUDIO_BIG_ENDIAN, or AUDIO_LITTLE_ENDIAN.

ALIGNMENT - Left / right data alignment in a sample, either AUDIO_ALIGNMENT_RIGHT or AUDIO_ALIGNMENT_LEFT.

BITS - Data depth in bits.

SIZE - Data size in bytes.

CHANNELS - Number of channels.

27.9 Error Codes

The I2S device driver defines the following error codes.

Table 27-3. I2S device driver error codes

Error code	Description
I2S_OK	Success.
I2S_ERROR_INVALID_PARAMETER	Initialization struct pointer is NULL.
I2S_ERROR_CHANNEL_INVALID	Selected channel is not available (>1).
I2S_ERROR_MODE_INVALID	MODE does not match I2S_MODE_SLAVE or I2S_MODE_MASTER.
I2S_ERROR_WORD_LENGTH_UNSUPPORTED	Invalid data word length.
I2S_ERROR_CLK_INVALID	Invalid clock source selected.
I2S_ERROR_BUFFER_SMALL	Buffer size is too small (<2).
AUDIO_ERROR_INVALID_IO_FORMAT	Invalid data format.

Chapter 28
HWTIMER Driver

28.1 Overview

This chapter describes the HWTIMER driver framework which provides a common interface for various timer modules.

The driver consists of two layers:

- Hardware specific lower layer contains implementation specifics for particular timer module. This layer is not intended for use by an application.
- Generic upper layer provides an abstraction to call the proper lower layer functions while passing a proper context structure to them. This chapter describes the generic upper layer only.

28.2 Source Code Location

The source code for HWTIMER drivers is located in `source\io\hwtimer` directory.

28.3 Header Files

To use HWTIMER driver, include the *hwtimer.h* and the device-specific *hwtimer_xxxx.h* header files from `source\io\hwtimer` in your application or in the BSP file *bsp.h*.

28.4 API Function Reference

All API functions take a pointer to caller allocated HWTIMER structure keeping the context necessary for the driver. This structure is opaque to the caller. The main purpose of the upper layer API is to provide the abstraction of the hardware specific lower layer driver.

28.4.1 hwtimer_init()

Synopsis

```
_mqx_int hwtimer_init
(
    HWTIMER_PTR hwtimer,
    const HWTIMER_DEVIF_STRUCT_PTR devif,
    uint32_t id,
    uint32_t int_priority
)
```

Parameters

hwtimer [out] — Pointer to hwtimer structure.

devif [in] — Pointer to a structure determining the lower layer.

id [in] — Numerical identifier of the timer within one timer module.

int_priority [in] — Interrupt priority.

Return Value

- MQX_OK (success)
- Error - Otherwise

Description

This function initializes caller allocated structure according to given parameters.

The device interface pointer determines low layer driver to be used. Device interface structure is exported by each low layer driver and is opaque to the applications. For details, refer to the chapter about the low layer driver below.

The meaning of the numerical identifier varies depending on the low layer driver used. Typically, it identifies a particular timer channel to initialize.

The initialization function has to be called prior to using any other HWTIMER driver API function.

28.4.2 hwtimer_deinit()

Synopsis

```
_mqx_int hwtimer_deinit
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

Return Value

- MQX_OK (De-initialization successful)
- Error - Otherwise

Description

This function calls lower layer de-initialization function and afterwards invalidates hwtimer structure by clearing it.

28.4.3 hwtimer_set_freq()

Synopsis

```
_mqx_int hwtimer_set_freq
(
    HWTIMER_PTR hwtimer,
    uint32_t    clock_id,
    uint32_t    freq
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

clock_id [in] — Clock identifier used for obtaining timer's source clock.

freq [in] — Required frequency of the timer in Hz.

Return Value

- MQX_OK (Setting frequency successful)
- Error - Otherwise

Description

This function configures the timer to tick at a frequency as closely as possible to the requested one. Actual accuracy depends on the timer module.

The function gets the value of the base frequency of the timer via the clock manager, calculates required divider ratio, and calls the low layer driver to set up the timer accordingly.

A call to this function might be consuming the CPU time as it may require complex calculation to choose the best configuration of dividers. The actual complexity depends on timer module implementation. Typically, if there is only single divider or counter preload value, there is no significant overhead.

28.4.4 hwtimer_get_freq()

Synopsis

```
uint32_t hwtimer_get_freq
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

Return Value

- Actual frequency in Hz.
- 0 - When an error occurs.

Description

The function returns the current frequency of the timer calculated from the base frequency and actual divider settings of the timer, or, if there is an error, it returns a zero.

28.4.5 hwtimer_set_period()

Synopsis

```
_mqx_int hwtimer_set_period
(
    HWTIMER_PTR hwtimer,
    uint32_t    clock_id,
    uint32_t    period
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

clock_id [in] — Clock identifier used for obtaining timer's source clock.

period [in] — Required period of the timer in us.

Return Value

- MQX_OK (setting period succeeded)
- Error - Otherwise

Description

This function provides an alternate way to set up the timer to a desired period specified in microseconds rather than to a frequency in Hertz. The function gets the value of the base frequency of the timer via the clock manager, calculates required divider ratio, and calls the low layer driver to set up the timer accordingly.

A call to this function might be consuming the CPU time as it may require complex calculation to choose the best configuration of dividers. The actual complexity depends on the timer module implementation. Typically, if there is only a single divider or a counter preload value, there is no significant overhead.

28.4.6 hwtimer_get_period()

Synopsis

```
uint32_t hwtimer_get_period
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

Return Value

- Actual period in micro seconds.
- 0 - When an error occurs.

Description

This function returns the current period of the timer in microseconds, which is calculated from the base frequency, and actual divider settings of the timer.

28.4.7 hwtimer_get_modulo()

Synopsis

```
uint32_t hwtimer_get_modulo
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

Return Value

- Actual resolution (modulo) of timer.
- 0 - When an error occurs.

Description

This function returns the period of the timer in sub-ticks. It is typically called after hwtimer_set_freq() or hwtimer_set_period() to obtain actual resolution of the timer in the current configuration.

28.4.8 hwtimer_start()

Synopsis

```
_mqx_int hwtimer_start
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to hwtimer structure.

Return Value

- MQX_OK (Hwtimer start successful)
- Error - Otherwise

Description

This function enables the timer and gets it running. The timer starts counting and generating interrupts each time it rolls over.

28.4.9 hwtimer_stop()

Synopsis

```
Synopsis
_mqx_int hwtimer_stop
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

Return Value

- MQX_OK (Hwtimer stop succeeded)
- Error - Otherwise

Description

The timer stops counting after this function is called. Pending interrupts and callbacks are canceled.

28.4.10 hwtimer_get_time()

Synopsis

```
_mqx_int hwtimer_get_time
(
    HWTIMER_PTR      hwtimer,
    HWTIMER_TIME_PTR time
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

time [out] — Returns current value of the timer.

Return Value

- MQX_OK (Getting time succeeded)
- Error - Otherwise

Description

This function reads the current value of the timer. Elapsed periods (ticks) and current value of the timer counter (sub-ticks) are filed into the HWTIMER_TIME structure. The sub-ticks number always counts up and is reset to zero when the timer overflows regardless of the counting direction of the underlying device. The returned value corresponds to lower 32 bits of the elapsed periods (ticks).

28.4.11 hwtimer_get_ticks()

Synopsis

```
uint32_t hwtimer_get_ticks
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

Return Value

- Low 32 bits of 64 bit tick value.
- 0 - When error occurs.

Description

This function returns lower 32 bits of elapsed periods (ticks). The value is guaranteed to be obtained automatically without needing to mask the timer interrupt. The lower layer driver is not involved at all, thus a call to this function is considerably faster than hwtimer_get_time().

28.4.12 hwtimer_callback_reg()

Synopsis

```
_mqx_int hwtimer_callback_reg
(
    HWTIMER_PTR           hwtimer,
    HWTIMER_CALLBACK_FPTR callback_func,
    void                 *callback_data
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

callback_func [in] — Function pointer to be called when the timer expires.

callback_data [in] — Arbitrary pointer passed as parameter to the callback function.

Return Value

- MQX_OK (callback registration succeeded)
- Error - Otherwise

Description

This function registers function to be called when the timer expires. The callback_data is arbitrary pointer passed as parameter to the callback function. This function must not be called from a callback routine.

28.4.13 hwtimer_callback_block()

Synopsis

```
_mqx_int hwtimer_callback_block
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

Return Value

- MQX_OK (Callback blocking succeeded)
- Error - Otherwise

Description

This function is used to block callbacks when execution of the callback function is undesired. If the timer overflows when callbacks are blocked, the callback becomes pending.

28.4.14 hwtimer_callback_unblock()

Synopsis

```
_mqx_int hwtimer_callback_unblock
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

Return Value

- MQX_OK (Callback unblocking succeeded)
- Error - Otherwise

Description

This function is used to unblock previously blocked callbacks. If there is a callback pending, it gets immediately executed. This function must not be called from a callback routine. It does not make sense to do so anyway since a callback function never gets executed while callbacks are blocked.

28.4.15 hwtimer_callback_cancel()

Synopsis

```
_mqx_int hwtimer_callback_cancel
(
    HWTIMER_PTR hwtimer
)
```

Parameters

hwtimer [in] — Pointer to a hwtimer structure.

Return Value

- MQX_OK (callback cancellation succeeded)
- Error - Otherwise

Description

This function cancels pending callback, if any.

28.5 Data Types Used by the HWTIMER API

The following data types are used within the HWTIMER driver.

28.5.1 HWTIMER

The context structure contains a pointer to a device interface structure, pointers to a callback function and its context, and private storage locations for arbitrary data keeping the context of the lower layer driver. The context structure is passed to all API functions except the other parameters. The application should not access members of this structure directly.

28.5.2 HWTIMER_DEVIF_STRUCT

Each low layer driver exports an instance of this structure initialized with pointers to API functions which the driver implements. The functions should be declared as static meaning that they are not exported directly.

28.5.3 HWTIMER_TIME_STRUCT

The hwtimer time structure represents a timestamp consisting of timer elapsed periods (TICKS) and current value of the timer counter (SUBTICKS).

Synopsis

```
typedef struct hwtimer_time_struct
{
    uint64_t TICKS;
    uint32_t SUBTICKS;
} HWTIMER_TIME_STRUCT, * HWTIMER_TIME_PTR;
```

Parameters

TICKS - Ticks of timer.

SUBTICKS - Subticks of timer.

28.6 Low Level Drivers Specifications

This chapter describes features related to various low level driver implementation. Currently only PIT timer module is supported. The implementation is extended to other timer modules in the upcoming MQX releases.

28.6.1 PIT

Configuration parameters:

- **BSPCFG_HWTIMER_PIT_FREEZE** - Allows the timers to be stopped when the device enters the Debug mode. Place this configuration into the *user_config.h*. if you require this functionality of the HWTIMER driver.

28.7 Example

The example for the HWTIMER driver that shows how to use HWTIMER driver API functions is provided with the MQX installation and is located in `mqx/examples/hwtimer` directory.

There are definitions in the BSP specific header file which provide the low level device structure, BSP_HWTIMER1_DEV, with id, BSP_HWTIMER1_ID, and input frequency for the timer module, BSP_HWTIMER1_SOURCE_CLK.

Chapter 29
FTM Quadrature Decoder Driver

29.1 Overview

This section contains information for the FTM Quadrature Decoder driver that accompany with Freescale MQX RTOS.

29.2 Location of Source Code

The source codes of FTM driver reside source/io/ftm/.

29.3 Header Files

Table 29-1. Header Files

Driver	Header file
FTM driver	ftm.h
FTM hardware-specific driver	ftm_quaddec.h

29.4 Installing FTM Quadrature Decoder Drivers

The sections below describe the processes for installing the FTM Quadrature Decoder drivers.

29.4.1 _frd_ftm_quaddec_install

Synopsis:

```
uint32_t _frd_ftm_quaddec_install(char * identifier, QUADDEC_INIT_INFO_STRUCT_PTR
init_data_ptr);
```

Parameters:

- Identifier [in] - A string that identifies the device for fopen
- Init_data_ptr [in] - The I/O init data pointer

29.4.2 Init Data

User should configure the init data before compiling. All init data is organized into the one structure type and all init data is pushed to FTM driver with install APIs.

Init data structure type:

```
typedef struct quaddec_init_info {
    /*
* FTM channel number
         * 0: FTM0
         * 1: FTM1
         * 2: FTM2
         * 3: FTM3
         */
    uint8_t          CHANNEL;
    /*
         * Indicates which clock source is selected, the clock source types are defined by
         * FTM_CLOCK_TYPE enum.
         * 1: System clock (IPG clock)
         * 2: Fixed frequency clock, depend on the clock controller settings
         * 3: external clock
         */
    uint8_t          CLK_TYPE;
    /*
         * Quadrature Decoder mode, refer to QUADDEC_MODE defines:
         */
    uint8_t          QUADDEC_MODE;
    /* Phase A input polarity, refer to QUADDEC_POL_TYPE defines */
    uint8_t          PHA_POL;
    /* Phase B input polarity, refer to QUADDEC_POL_TYPE defines */
    uint8_t          PHB_POL;
    /* Filter value, this filed is useful only when filter is enabled */
    uint8_t          FILTER_VAL;
    /* the value of modulo register*/
    uint16_t         MODULO;
    /* the initial value counter, this value is used to reset the COUNTER register */
    uint16_t         CNT_INIT;
    /* Flag indicates enabling or disabling FTM hardware filter */
    bool             FILTER_EN;
    /*
         * Enable FTM Quaddec interrupt or not, if enabled,
         * driver works on interrupt mode, if not, works on
         * pooling mode */
    bool            INT_EN;
} QUADDEC_INIT_INFO_STRUCT, _PTR_ QUADDEC_INIT_INFO_STRUCT_PTR;
```

Each Quadrature driver has its own init data, it is a structure variable. All these init data are configured before compiling, and can't be updated at runtime.

29.5 I/O Control Commands

This section describes the I/O control commands that you use when you call **ioctl.** They are defined in *ftm.h*.

Table 29-2. I/O control commands

Command	Description	Parameters
IO_IOCTL_FTM_GET_CHANNEL	Get the FTM channel number.	uint32_t *
IO_IOCTL_FTM_REG_DUMP	Dump FTM register	None(NULL)
IO_IOCTL_FTM_QUADDEC_SET_CB	Register CB to FTM Quadrature Decoder driver, it is only used when driver works on async inquiring mode. Refer to ***FTM_QUADDEC_CALLBACK_FUNCTION_PTR*** for the callback function prototype.	FTM_QUADDEC_CALLBACK_STRUCT_PTR
IO_IOCTL_FTM_QUADDEC_GET_EVENT	Get FTM Quadrature Decoder event. It is only used when driver works on sync inquiring mode. Refer to ***FTM_QUADDEC_EVENT*** for the event definitions.	uint32_t *
IO_IOCTL_FTM_QUADDEC_GET_MODE	Get FTM Quadrature Decoder inquiring mode. Refer to ***QUADDEC_INQUIRE_MODE*** for the inquiring mode definitions.	uint32_t*

29.6 Inquiring mode

FTM Quadrature Decoder driver provides two kind of inquiring mode:

1. Synchronous mode

   If FTM Quadrature Decoder driver works on this mode, application can get driver event by calling **IOCTL** with ***IO_IOCTL_FTM_QUADDEC_GET_MODE*** command.

2. Asynchronous mode

   If FMT Quadrature Decoder driver works on this mode, application should register its callback function to driver by calling ***IOCTL*** with ***IO_IOCTL_FTM_QUADDEC_SET_CB***. The driver events are sent to application by this callback function.

Note

the application callback function is called by FTM Quadrature Driver ISR and works in MQX interrupt context.

29.7 Example

For basic use, see MQX RTOS examples — FTM example in directory *mqx/examples/ftm*.

29.8 Error Codes

The FTM Quadrature Decoder driver only uses the MQX I/O error codes.

Chapter 30
I/O Expander Driver

30.1 Overview

I/O Expander Driver aims to control an off-chip I/O expander device and provides convenient interfaces for users to handle each pin. The driver is divided into two parts:

Hardware-independent generic driver

Hardware-dependent layer called hardware-specific driver, currently only support the MAX7310 device.

30.2 Location of Source Code

Table 30-1. Source Code

Driver	Location
I/O Expander driver	source/io/io_expander

30.3 Header Files

Table 30-2. Header Files

Driver	Header file
I/O Expander driver	io_expander.h
I/O Expander hardware-specific driver	io_exp_<device_name>.h

30.4 Installing Drivers

Each I/O expander 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/O expander hardware modules.

To install the MAX7310 I/O Expander driver for a device, the installing function **_max7310_install()** should be used.

Example of the _max7310_install() function call

```
#if BSPCFG_ENABLE_IO_EXPANDER_MAX7310
    _max7310_install("ioexp0:", &_bsp_max7310_init);
#endif
```

30.5 Initialization Records

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

Synopsis for MAX7310 device

```
typedef struct io_exp_max7310_init_struct
{
   uint8_t                 DEV_ID;
   uint8_t                 I2C_address;
   const char*             I2C_control;
} IOEXP_MAX7310_INIT_STRUCT, * IOEXP_MAX7310_INIT_STRUCT_PTR;
```

Parameters

- DEV_ID - The device ID to identify different MAX7310 device
- I2C_address - The I2C address for this MAX7310 I/O expander
- I2C_control - The I2C bus name the MAX7310 locates on

Example

The following is an example for the MAX7310 as it can be found in the appropriate BSP code (for example the *init_ioexp.c* file)

```
const IOEXP_MAX7310_INIT_STRUCT _bsp_max7310_init = {
    0,
    0x18,
    "i2c1:"
};
```

30.6 Driver Services

The I/O expander device driver provides these services:

Table 30-3. Driver Services

API	Calls	Description
_io_fopen()	_io_expander_open()	Open an I/O Expander device driver. Multiple opens to a device is supported. Each returned file handler need to bind a specific pin before it begins other operations.
_io_fclose()	_io_expander_close()	This closes the device driver for current user. If the device is shared by other users, the close won't affect others.
_io_ioctl()	_io_expander_ioctl()	IOCTL functions. Referring to the IOCTL Command table for detail. Note that the application should check if an error was returned from the IOCTL call and handle it accordingly.
_io_read()	_io_expander_read()	Read the logic value from an input I/O pin on the device.
_io_write()	_io_expander_write()	Write the logic value to an output I/O pin on the device.

30.7 Generic IOCTL Commands

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

Table 30-4. Control Commands

Command	Description	Parameters
IO_IOCTL_IOEXP_SET_PIN_NO	Set the I/O pin number which is ready for use	*param_ptr* - pointer to uint32_t
IO_IOCTL_IOEXP_SET_PIN_DIR_IN	Configure the I/O pin to a input pin	*param_ptr* - pointer to uint32_t
IO_IOCTL_IOEXP_SET_PIN_DIR_OUT	Configure the I/O pin to a output pin	*param_ptr* - pointer to uint32_t
IO_IOCTL_IOEXP_SET_PIN_VAL_HIGH	Set logic level of a output pin to high level	none (NULL)
IO_IOCTL_IOEXP_SET_PIN_VAL_LOW	Set logic level of a output pin to low level	none (NULL)
IO_IOCTL_IOEXP_GET_INPUT_REG	Get the logic value of all pins, regardless of the pins direction.	*param_ptr* - pointer to uint32_t
IO_IOCTL_IOEXP_CLEAR_PIN_NO	Release owned pin. After it, the file handler turns to have invalid pin number.	none (NULL)

The following is an example of using IOCTL command for the I/O expander driver:

Set the pin number 1 be bound to the file handler:

```
ioctl(file, IO_IOCTL_IOEXP_SET_PIN_NO, (uint32_t* 1);
```

Set the pin direction to be output:

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

Set the pin value to logic high level:

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

30.8 Hardware-Specific IOCTL Commands

Hardware-specific commands are used to handle specific I/O expander device behavior and hardware performance. These commands are not portable to other I/O expander device.

No hardware-specific commands are implemented yet.

30.9 Error Codes

No additional error codes are generated.

Chapter 31
DMA Driver Framework

31.1 Overview

The DMA driver is a plain C driver providing an abstraction of the DMA engine to hide its complexity and differences in the implementation on various platforms.

31.2 Location of source code

The source code for DMA driver framework is located in *source/io/dma*.

31.3 Header files

To use DMA driver include header *dma.h* in your application or in the BSP file *bsp.h*.

31.4 Internal design of DMA drivers

The DMA driver framework features a layered design with two distinct layers which includes the low level drivers and a generic layer on top of the drivers.The low level drivers are device specific and implement the necessary hardware abstraction function sets. The generic layer is device independent and provides the common API for low level driver and generic functionality.

31.5 Design the API

The API covers the DMA engine initialization, channel and transfer setup, enabling and disabling request source and a support for hooking up a completion callback routine (executed from ISR).

Virtual channels

Each channel, which is handled by any of the registered low level drivers, is assigned a unique virtual channel number. Virtual channel numbers are assigned according to the BSP provided DMA device interface list.

Channel handles

Prior to using a channel, the application has to obtain a channel handle by claiming one of the available virtual channels. The same virtual channel cannot be claimed the second time without releasing the handle first.

All subsequent operations on the channel are performed using the channel handle.

Transfer control descriptors

The data structure which contains the information necessary to set up a data transfer is referred to as a transfer control descriptor or TCD. Note that the TCD is a generic data structure defined by the API, not the TCD used by eDMA engine.

To avoid confusion when the context is unclear, the TCD is referred to as a **software TCD**, as opposed to the **hardware TCD** used by the DMA engine. While hardware TCDs may be different, depending on the type or version of the DMA engine, the software TCD is well defined by the DMA API and provides an abstraction.

The software TCD contains numerous parameters which define transfer and its properties, such as source/destination address, data length, transfer width, etc. However, not all combinations of parameter are necessarily supported by all DMA drivers/modules.

Main reasons for using the software TCDs are:

- It is impractical to pass all parameters needed to set up a transfer as parameters to a function call.
- TCD may be easily extended to cover functionality introduced by the future DMA modules without breaking the compatibility at the function call level.
- Generic functions, which are used to prepare TCDs for common types of transfers, are provided for ease of use.

Transfer descriptor chaining

To support transfer chaining for continuous data stream, the DMA driver needs memory slots for parameters of chained (waiting) transfers (TCD queue). For eDMA, the parameters are kept as images of hardware TCDs which are directly fetched from the memory by the DMA engine. A similar approach could likely be used with other DMA modules.

The low level driver allocates the properly aligned memory area for TCD slots according to the application requirements during the initial channel setup.

For continuous data streaming it is often sufficient to request only two slots which are used as two-depth queue (one transfer in progress and one waiting). The callback completion function is executed after the slot, which corresponds to a recently finished transfer, is freed. Therefore, all subsequent transfers may be submitted directly in the completion callback function.

The channel may also be configured in a loop mode. In this mode, the complete transfer TCDs are not freed up or removed from queue. When the queue becomes full, the first TCD is chained to the last one and forms an endless loop. An example of this mode involves generating periodical signals by transferring samples to DAC. The only way to break the loop transfer is to reset the channel or set it up again.

Request sources

To transfer data, the DMA channel requires a request signal indicating that a transfer is about to occur. The DMA channel is typically associated with a particular peripheral by selecting a source of the request signal.

A transfer on a channel takes place when all of these conditions are met:

- A request for the channel is enabled.
- The selected request signal source is active (peripheral requests a transfer).
- There is an unfinished or a pending TCD in the queue.

When there is no TCD in the queue, any transfer on the channel is suspended although the request might be enabled and the selected request signal source is active. The transfer resumes automatically once a valid TCD is submitted, unless the request is disabled or a request signal becomes inactive.

31.6 Summary of the API

Supported API calls controlling the DMA engine:

- dma_init
- dma_deinit
- dma_channel_claim
- dma_channel_release
- dma_channel_reset
- dma_channel_setup
- dma_channel_status
- dma_transfer_submit
- dma_request_source
- dma_request_enable
- dma_request_disable
- dma_callback_reg

Supported generic API calls for handling software TCDs:

- dma_tcd_memcpy
- dma_tcd_mem2reg
- dma_tcd_reg2mem

Data structures

- typedef struct dma_tcd {
- uint32_t SRC_ADDR;
- uint32_t SRC_WIDTH;
- int32_t SRC_OFFSET;
- uint32_t SRC_MODULO;
- uint32_t DST_ADDR;
- uint32_t DST_WIDTH;
- int32_t DST_OFFSET;
- uint32_t DST_MODULO;

- uint32_t LOOP_BYTES;
- uint32_t LOOP_COUNT;
- int32_t LOOP_SRC_OFFSET;
- int32_t LOOP_DST_OFFSET;
- } DMA_TCD;
- SRC_ADDR — starting source address of the transfer. The requirements for alignment of the address value are specific to a particular DMA engine.
- SRC_WIDTH — width of a single transfer in bytes. Supported width values are specific to a particular DMA engine.
- SRC_OFFSET — signed value added to the source address after each transfer. The only value guaranteed to be supported is the same as SRC_WIDTH, which is also used if zero is specified.
- SRC_MODULO — defines the range of a circular buffer. Supported values are specific to a particular DMA driver.
- DST_ADDR — see SRC_ADDR
- DST_WIDTH — see SRC_WIDTH
- DST_OFFSET — see SRC_OFFSET
- DST_MODULO — see SRC_MODULO
- LOOP_BYTES — defines the number of bytes transferred upon data request. The value should be properly aligned with the SRC_WIDTH and the DST_WIDTH.
- LOOP_COUNT — a number of the loop iteration. The total amount of data to be transferred is LOOP_BYTES*LOOP_COUNT.
- LOOP_SRC_OFFSET — a signed value used to adjust an address after each loop.
- LOOP_DST_OFFSET — see LOOP_SRC_OFFSET. The DMA engine may not support the usage of the LOOP_SRC_OFFSET and LOOP_DST _OFFSET.

31.7 Functional description of the API

DMA framework initialization

```
int dma_init(
const DMA_DEVIF_LIST *init_devif_list
);
```

The function initializes the DMA driver framework. The structure passed to this function defines a list of the low level drivers which handle particular channels that get initialized during the function execution.

DMA framework deinitialization

```
int dma_deinit(void);
```

The function deinitializes the DMA driver framework including all previously initialized low level drivers.

Claiming a channel

```
        int dma_channel_claim(
DMA_CHANNEL_HANDLE *channel,
int vchannel
);
```

This function is used to claim a particular channel from the DMA framework. If the channel, identified by virtual channel number, is available, the channel handle is filled in and success is reported. Otherwise, an error status is returned. If the claim operation succeeds, the channel becomes unavailable for subsequent claim operations until it is released using the dma_channel_release. This function is guaranteed to be thread-safe, which means that the claim operation is atomic.

Releasing previously claimed channel

```
    int dma_channel_release(
DMA_CHANNEL_HANDLE *channel
);
```

This function is used to release a previously claimed channel. The channel, then, becomes available for subsequent claims.

Resetting a channel

```
        int dma_channel_reset(
DMA_CHANNEL_HANDLE *channel
);
```

This function brings the channel to the initial state by canceling pending transfers or transfers in progress. It is also used to recover the channel from an error state.

Channel setup

```
int dma_channel_setup(
DMA_CHANNEL_HANDLE channel,
int tcds_slots,
uint32_t flags
);
```

This function sets up the DMA channel transfer parameters. The driver prepares the queue slots for waiting TCDs according to the tcd_slots parameter. A value of zero or one means that no transfer chaining is required and only one TCD may be submitted at one time.

Additional flags may be passed to define special requirements:

- DMA_CHANNEL_FLAG_LOOP_MODE — the descriptor submitted to the channel is chained in a loop.

Obtaining channel status

```
int dma_channel_status(
DMA_CHANNEL_HANDLE channel,
uint32_t *tcd_seq,
uint32_t *tcd_remaining
);
```

This function provides information about a momentary status of a transfer on the given channel. The values, which are returned via pointers tcd_seq and tcd_remaining, indicate a progress of the transfer. The progress of a transfer involved a sequence number of the current TCD and bytes which are yet to be transferred. If no transfer is submitted to the channel, both returned values are zero. Otherwise, a zero value returned via tcd_remaining indicates that the TCD identified by the tcd_seq has already finished the transfer.

Submitting a transfer

```
int dma_transfer_submit(
DMA_CHANNEL_HANDLE channel,
DMA_TCD *tcd,
uint32_t *tcd_seq
);
```

The function prepares a transfer on a given channel according to the parameters in the DMA_TCD and places it into the queue. If the DMA module does not support the combination of parameters in the TCD, the function fails and returns an error without touching the transfer queue. If the operation succeeds, the sequence number assigned to the transfer is optionally returned via tcd_seq_pointer.

If the channel is configured in a loop mode and the TCD is the last one according to the number of slots specified in a call to a dma_channel_setup, the loop is automatically closed by chaining to the TCD which was submitted as the first.

Selecting DMA request source

```
ng DMA request source
int dma_request_source(
DMA_CHANNEL_HANDLE channel,
uint32_t source
);
```

The function is used to select a source for a given channel. The source is an opaque identifier which is passed to the low level driver. The caller is responsible for selecting a proper request source according to the purpose of the channel.

Enabling DMA request for given channel

```
int dma_request_enable(DMA_CHANNEL_HANDLE channel);
```

The function enables the DMA request for a given channel from a configured source. The data flow is controlled by the source after the execution of the function. If a TCD is pending, the transfer is resumed automatically.

Disabling DMA request for given channel

```
int dma_request_disable(DMA_CHANNEL_HANDLE channel);
```

The function disables the DMA request for a given channel. A transfer in progress is suspended until the request is re-enabled.

Registration of callback function

```
int dma_callback_reg(
DMA_CHANNEL_HANDLE channel,
DMA_EOT_CALLBACK callback_func,
void *callback_data
);
```

This is a registers function which is used when there is an event on the channel, either a transfer is finished after the last iterations of the transfer loop or there is a transfer error. The callback function is executed in the context of the interrupt service routine.

```
typedef void (_CODE_PTR_ DMA_EOT_CALLBACK)(
void *callback_data,
int tcds_done,
uint32_t tcd_seq
);
```

The callback function is passed as an arbitrary pointer which is specified when registering the callback to keep the context data for the callback function.

Prior to the execution of the callback function, the finished TCDs are removed from the queue. The callback function is passed, the number of finished TCDs removed from the queue (tcds_done), and the sequence number of the first finished/removed TCD (tcd_seq).

Zero or negative value of tcds_done indicates a transfer error. When this happens, the tcd_seq contains a sequence number of the last loaded TCD. The state of the TCD is undefined.

The only way to bring the channel into a defined state after an error is to perform a channel reset.

31.8 Helper functions for TCD handling

Preparing memory to memory TCD

```
void dma_tcd_memcpy(
DMA_TCD *tcd,
void *src,
void *dst,
uint_32 size
);
```

The function prepares TCD for a memory-to-memory copy. The function automatically selects the best DMA channel width for the best utilization of the bus bandwidth (the largest one with respect to the src/dst/size alignment).

Preparing memory to register TCD

```
int dma_tcd_mem2reg(
DMA_TCD *tcd,
volatile void *reg,
int regw,
void *src,
uint_32 size
);
```

The function prepares the TCD for a memory-to-register transfer. The largest possible transfer width with respect to src/size alignment is used on memory side while register width (regw parameter) is always respected. If negative number is passed in the regw, the TCD is prepared to that byte endianess and swapping is performed during the transfer.

Preparing register to memory TCD

```
int dma_tcd_reg2mem(
DMA_TCD *tcd,
volatile void *reg,
int regw, void *dst,
uint_32 size
);
```

The function prepares TCD for register-to-memory transfer. The largest possible transfer width with respect to dst/size alignment is used on the memory side while register width (regw parameter) is always respected. If a negative number is passed in the regw, the TCD is prepared to that byte endianess and swapping is performed during the transfer.

31.9 Usage Scenarios

Plain FIFO

Application claims a channel, configures it by using dma_channel_setup and selects a request source by the dma_request_source. Let's assume the selected request source is active and the application calls the dma_request_enable on the channel. The actual transfer does not start at this point because there is no descriptor submitted to the channel.

Once a valid descriptor (1) is submitted using the dma_transfer_submit function the descriptor is placed to the head of the TCD queue and transfer starts immediately.

If another descriptor (2) is submitted using the dma_transfer_submit, while the previous transfer is still in progress, it is placed in the queue and chained to the descriptor 1.

When descriptor 1 finishes, descriptor 2 is loaded automatically and starts executing. Interrupt is then invoked during which the descriptor 1 is removed from the queue, causing descriptor 2 to become the queue head, at which point, the callback routine gets executed.

When the last descriptor in the queue finishes, the transfer is suspended until another descriptor is submitted.

Scatter/gather:

This scenario is similar to FIFO. The only difference is that the application does not call the dma_request_enable to enable channel's request source but rather submits a descriptor first. The descriptor is placed in the queue but transfer does not start because the request source is not enabled.

There can be as many descriptors submitted as the queue length, specified in the dma_channel_setup, allows. The submitted descriptors are placed into the queue and chained one after another.

Once the dma_request_enable is called, the first descriptor in the queue (head) is started and the process continues in a similar fashion as with the FIFO mode.

This way a scatter/gather list may be prepared first and then transferred without a disruption in one go.

After all descriptors are processed, the application can call the dma_request_disable and repeat the whole process from the beginning.

Looping over multiple buffers

The application calls dma_channel_setup with a flag specifying the loop mode. In this case, the finished descriptors are not removed from the queue when they are finished.

Let's assume that the application specified the queue length for 3 descriptors during a call to the dma_channel_setup. After the descriptor 3 is submitted and the queue becomes full, the loop is automatically closed by chaining from descriptor 3 to descriptor 1.

The point at which the DMA request is enabled determines when the transfer starts but does not affect the described principle of looping itself.

Note that, in the loop mode, the descriptor sequence numbers reported by the dma_channel_setup do not increment indefinitely as the loop runs but they correspond to a number of the slot in the queue. In the described case of the 3 descriptors the sequence would be: 0, 1, 2, 0, 1, 2, 0... etc.

Chapter 32
ASRC Driver Framework

32.1 Overview

The ASRC driver can be divided into a generic layer and a hardware-specific layer. The ASRC driver implements the standard MQX I/O subsystem API interfaces.

32.2 Location of source code

The source code for ASRC driver framework is located in *source/io/asrc*.

32.3 Header files

To use the ASRC driver, include the header *asrc.h* in your application. For ASRC hardware specific driver, use *asrc_vybrid.h*.

32.4 Installing drivers

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

Synopsis for ASRC in Vybrid

```
typedef struct vybrid_asrc_init_struct{
     /* The device ID to identify different device*/
     uint8_t                  DEV_ID;
     /* The channel's number for ASRC pair A, this value can't be changed dynamically*/
     uint8_t                  PAIR_A_CHANNELS;
     /* The channel's number for ASRC pair B, this value can't be changed dynamically*/
     uint8_t                  PAIR_B_CHANNELS;
     /* The channel's number for ASRC pair C, this value can't be changed dynamically*/
     uint8_t                  PAIR_C_CHANNELS;
     /* Default slot width for input port*/
     uint8_t                  INPUT_SLOT_WIDTH;
     /* Default slot width for output port*/
     uint8_t                  OUTPUT_SLOT_WIDTH;
     } VYBRID_ASRC_INIT_STRUCT, * VYBRID_ASRC_INIT_STRUCT_PTR;
```

Example

```
const VYBRID_ASRC_INIT_STRUCT _bsp_asrc_init
=
{

0,

2,

2,
    2,

32,

32
};
```

Example of the _asrc_vybrid_install() function call

```
#if BSPCFG_ENABLE_ASRC
        _asrc_vybrid_install("asrc0:", &_bsp_asrc_init);
#endif
```

32.5 Driver Services

Table 32-1. Driver services

API	Calls	Description
_io_fopen()	_io_asrc_open()	Open an ASRC device driver. A device can be opened multiple times. Each returned file handler is needed to bind an ASRC pair.
_io_fclose()	_io_asrc_close()	This closes the device driver for a current user. If the device is shared by other users, the close by one user does not affect others.
_io_ioctl()	_io_asrc_ioctl()	IOCTL functions. See the the IOCTL command table for details. The application should check if an error was returned from the IOCTL call and handle it accordingly.
_io_uninstall()	_io_asrc_uninstall ()	Uninstall ASRC driver.

32.6 I/O Control Commands

Table 32-2. I/O control commands

Command	Description	Parameters
IO_IOCTL_ASRC_CONFIG	Applies configuration to the ASRC set.	*param_ptr* - pointer to ASRC_SET_CONFIG_STRUCT
IO_IOCTL_ASRC_SET_SAMPLE_RATE	The command applies a sample rate setting to the ASRC set.	*param_ptr* - pointer to ASRC_SAMPLE_RATE_PAIR_STRUCT
IO_IOCTL_ASRC_START	The command starts the conversion of the ASRC set.	none (NULL)
IO_IOCTL_ASRC_STOP	The command stops the conversion of the ASRC set.	none (NULL)
IO_IOCTL_ASRC_SET_REFCLK	The command applies a reference clock setting to the ASRC set.	*param_ptr* - pointer to ASRC_REF_CLK_PAIR_STRUCT
IO_IOCTL_ASRC_SET_SLOT_WIDTH	The command applies a slot width setting to the ASRC set.	*param_ptr* - pointer to ASRC_CLK_DIV_PAIR_STRUCT
IO_IOCTL_ASRC_SET_IO_FORMAT	The command applies the FIFO format setting to the ASRC set.	*param_ptr* - pointer to ASRC_IO_FORMAT_PAIR_STRUCT
IO_IOCTL_ASRC_INSTALL_SERVICE_SRC	The command is used to install the DMA service on the input port of the ASRC set.	*param_ptr* - pointer to ASRC_INSTALL_SERVICE_STRUCT
IO_IOCTL_ASRC_INSTALL_SERVICE_DEST	The command is used to install the DMA service on the output port of the ASRC set.	*param_ptr* - pointer to ASRC_INSTALL_SERVICE_STRUCT
IO_IOCTL_ASRC_UNINSTALL_SRC	The command is used to uninstall the DMA service from the input port of the ASRC set.	none (NULL)
IO_IOCTL_ASRC_UNINSTALL_DEST	The command is used to uninstall the DMA service from the output port of the ASRC set.	none (NULL)

32.7 ASRC_SET_CONFIG_STRUCT

This section describes the ASRC_SET_CONFIG_STRUCT parameter, which can be passed whenever the _io_ioctl() is called with the IO_IOCTL_ASRC_CONFIG command.

Synopsis for ASRC_SET_CONFIG_STRUCT

```
typedef struct asrc_set_config
    {
    /*The input data's sample rate, unit: HZ*/
    uint32_t input_sample_rate;
    /*The target sample rate to output, unit: HZ*/
        uint32_t output_sample_rate;
    /*The reference clock for ASRC pair's input port*/
```

```
        ASRC_CLK input_ref_clk;
    /*The reference clock for ASRC pair's output port*/
        ASRC_CLK output_ref_clk;
        /*The FIFO word format for ASRC pair's input FIFO*/
        ASRC_FIFO_FORMAT input_fifo_fmt;
        /*The FIFO word format for ASRC pair's output FIFO*/
        ASRC_FIFO_FORMAT output_fifo_fmt;
    /*The slot width for ASRC pair's input port, it is used for clock divider,
    8, 16, 24, 32, 64*/
        uint8_t input_slot_width;
    /*The slot width for ASRC pair's output port, it is used for clock divider,
    8, 16, 24, 32, 64*/
        uint8_t output_slot_width;
        /*The threshold for ASRC pair's input FIFO*/
        uint8_t input_fifo_threshold;
        /*The threshold for ASRC pair's output FIFO*/
        uint8_t output_fifo_threshold;
}ASRC_SET_CONFIG_STRUCT, * ASRC_SET_CONFIG_STRUCT_PTR;
```

Synopsis for ASRC_CLK

```
typedef enum asrc_clk
    {
         /*Clock source from NULL*/
    ASRC_CLK_NONE = 0x0,
    /*Clock source from ESAI receiver */
    ASRC_CLK_ESAI_RX,
    /*Clock source from SAI0 receiver */
    ASRC_CLK_SAI0_RX,
    /*Clock source from SAI1 receiver */
    ASRC_CLK_SAI1_RX,
    /*Clock source from SAI2 receiver */
    ASRC_CLK_SAI2_RX,
    /*Clock source from SAI3 receiver */
    ASRC_CLK_SAI3_RX,
    /*Clock source from SSI1 receiver */
    ASRC_CLK_SSI1_RX,
    /*Clock source from SSI2 receiver */
    ASRC_CLK_SSI2_RX,
    /*Clock source from SSI3 receiver */
    ASRC_CLK_SSI3_RX,
    /*Clock source from SPDIF receiver */
    ASRC_CLK_SPDIF_RX,
    /*Clock source from MLK clock*/
    ASRC_CLK_MLB_CLK,
    /*Clock source from ESAI transmitter */
    ASRC_CLK_ESAI_TX,
    /*Clock source from SAI0 transmitter */
    ASRC_CLK_SAI0_TX,
    /*Clock source from SAI1 transmitter */
    ASRC_CLK_SAI1_TX,
    /*Clock source from SAI2 transmitter */
    ASRC_CLK_SAI2_TX,
    /*Clock source from SAI3 transmitter */
    ASRC_CLK_SAI3_TX,
    /*Clock source from SSI1 transmitter */
    ASRC_CLK_SSI1_TX,
    /*Clock source from SSI2 transmitter */
    ASRC_CLK_SSI2_TX,
    /*Clock source from SSI3 transmitter */
    ASRC_CLK_SSI3_TX,
    /*Clock source from SPDIF transmitter */
    ASRC_CLK_SPDIF_TX,
    /*Clock source from custom clock */
    ASRC_CLK_CUSTOM_CLK,
    /*Clock source from external audio clock */
    ASRC_CLK_EXT_AUD_CLK,
    /*Clock source from NULL */
```

```
    ASRC_CLK_NA
    }ASRC_CLK;
```

Synopsis for ASRC_FIFO_FORMAT

```
    typedef enum asrc_fifo_format
{
        /* LSB, valid bit is 8 */
        ASRC_FIFO_8BITS_LSB = 0x0,
        /* LSB, valid bit is 16 */
        ASRC_FIFO_16BITS_LSB,
        /* LSB, valid bit is 24 */
        ASRC_FIFO_24BITS_LSB,
        /* MSB, valid bit is 8 */
        ASRC_FIFO_8BITS_MSB,
        /* MSB, valid bit is 16 */
        ASRC_FIFO_16BITS_MSB,
        /* MSB, valid bit is 24 */
        ASRC_FIFO_24BITS_MSB,
        /* invalid format */
        ASRC_FIFO_FORMAT_NA
        }ASRC_FIFO_FORMAT;
```

32.8 ASRC_SAMPLE_RATE_PAIR_STRUCT

This section describes the ASRC_SAMPLE_RATE_PAIR_STRUCT parameter, which can be passed whenever the _io_ioctl() is called with the IO_IOCTL_ASRC_SET_SAMPLE_RATE command.

Synopsis for ASRC_SAMPLE_RATE_PAIR_STRUCT

```
typedef struct asrc_sample_rate_pair
    {
    /*The input data's sample rate to input to ASRC pair, unit: HZ*/
    uint32_t ASRC_INPUT_SAMPLE_RATE;
    /*The target sample rate to output from ASRC pair, unit: HZ*/
    uint32_t ASRC_OUTPUT_SAMPLE_RATE;
    } ASRC_SAMPLE_RATE_PAIR_STRUCT, * ASRC_SAMPLE_RATE_PAIR_STRUCT_PTR;
```

32.9 ASRC_REF_CLK_PAIR_STRUCT

This section describes the ASRC_REG_CLK_PAIR_STRUCT parameter, which can be passed whenever the _io_ioctl() is called with the IO_IOCTL_ASRC_SET_REFCLK command.

Synopsis for ASRC_REF_CLK_PAIR_STRUCT

```
typedef struct asrc_ref_clk_pair
    {
    /*The reference clock for ASRC pair's input port*/
    ASRC_CLK ASRC_INPUT_CLK;
    /*The reference clock for ASRC pair's output port*/
    ASRC_CLK ASRC_OUTPUT_CLK;
    } ASRC_REF_CLK_PAIR_STRUCT, * ASRC_REF_CLK_PAIR_STRUCT_PTR;
```

32.10 ASRC_CLK_DIV_PAIR_STRUCT

This section describes the ASRC_CLK_DIV_PAIR_STRUCT parameter, which can be passed whenever the _io_ioctl() is called with the IO_IOCTL_ASRC_SET_SLOT_WIDTH command.

Synopsis for ASRC_CLK_DIV_PAIR_STRUCT

```
typedef struct asrc_clk_div_pair
    {
    /*The slot width for ASRC pair's input port, it is used for clock divider,
    8, 16, 24, 32, 64*/
    uint8_t ASRC_INPUT_CLK_DIV;
    /*The slot width for ASRC pair's output port, it is used for clock divider,
    8, 16, 24, 32, 64*/
    uint8_t ASRC_OUTPUT_CLK_DIV;
    } ASRC_CLK_DIV_PAIR_STRUCT, * ASRC_CLK_DIV_PAIR_STRUCT_PTR;
```

32.11 ASRC_IO_FORMAT_PAIR_STRUCT

This section describes the ASRC_IO_FORMAT_PAIR_STRUCT parameter, which can be passed whenever the _io_ioctl() is called with the IO_IOCTL_ASRC_SET_IO_FORMAT command.

Synopsis for ASRC_IO_FORMAT_PAIR_STRUCT

```
typedef struct asrc_io_format_pair
    {
    /*The FIFO word format for ASRC pair's input FIFO*/
    ASRC_FIFO_FORMAT ASRC_INPUT_FORMAT;
    /*The FIFO word format for ASRC pair's output FIFO*/
    ASRC_FIFO_FORMAT ASRC_OUTPUT_FORMAT;
    } ASRC_IO_FORMAT_PAIR_STRUCT, * ASRC_IO_FORMAT_PAIR_STRUCT_PTR;
```

32.12 ASRC_INSTALL_SERVICE_STRUCT

This section describes the ASRC_INSTALL_SERVICE_STRUCT parameter, which can be passed whenever the _io_ioctl() with the IO_IOCTL_ASRC_INSTALL_SERVICE_SRC/ IO_IOCTL_ASRC_INSTALL_SERVICE_DEST with commands.

Synopsis for ASRC_INSTALL_SERVICE_STRUCT

```
typedef struct asrc_install_service
    {
    /*installed DMA service type*/
    ASRC_SERVICE_TYPE ASRC_SERVICE;
    /*The DMA channel id, returned from ASRC driver. User can use it to access
    DMA driver*/
```

```
    int32_t ASRC_DMA_CHL;
    } ASRC_INSTALL_SERVICE_STRUCT, * ASRC_INSTALL_SERVICE_STRUCT_PTR;
```

Synopsis for ASRC_SERVICE_TYPE

```
typedef enum asrc_service_type {
   /* service type is memory */
   ASRC_SERVICE_MEM = 0x0,
   /* service type is ESAI */
   ASRC_SERVICE_ESAI,
   /* service type is SAI0 */
   ASRC_SERVICE_SAI0,
   /* service type is SAI1 */
   ASRC_SERVICE_SAI1,
   /* service type is SAI2 */
   ASRC_SERVICE_SAI2,
   /* service type is SPDIF */
   ASRC_SERVICE_SPDIF,
   /* service type is SSI1 */
   ASRC_SERVICE_SSI1,
   /* service type is SSI2 */
   ASRC_SERVICE_SSI2,
   /* service type is SSI3 */
   ASRC_SERVICE_SSI3
  }ASRC_SERVICE_TYPE;
```

32.13 Error Codes

No additional error codes are generated.

Chapter 33
ESAI Driver Framework

33.1 Overview

This section describe ESAI driver in MQX RTOS. ESAI driver can be dived into generic layer and hardware specific layer. ESAI driver implement standard MQX I/O Subsystem interface.

33.2 Location of source code

The source code for ESAI driver framework is located in source/io/esai.

33.3 Header files

To use the ESAI driver, include the header *Vport/esai_vport.h* in your application. For ASRC hardware specific driver, use *Vport/esai_vport_vybrid.h*.

33.4 Installing drivers

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

To install the ESAI driver for a ESAI device two installing functions should be called.

Example of ESAI driver installation function call

```
esai_fifo_handle = _vybrid_esai_fifo_install(&_bsp_esai_init);
#if BSPCFG_ENABLE_ESAI_TX0
    _vybrid_esai_vport_install("esai_tx0:",
        &_bsp_esai_vport0_tx_init, esai_fifo_handle);
#endif
Synopsis for ESAI fifo device
typedef struct
{
/* The ESAI controller to initialize, for Vybrid, it only can be 0 */
uint32_t                MODULE_ID;
/*
* Transmitter FIFO water mark default value, unit: word(32 bts)
* The transmit FIFO empty flag will be set when the number of empty slots in the
* Transmit FIFO has met or exceeded the Transmit FIFO Watermark. In this case,
* ESAI Transmitter DMA request line is also driven.
* Refer to Vybrid reference manual for details.
*/
uint8_t                 TX_FIFO_WM;
/*
* Receiver FIFO water mark default value, unit: word(32 bts)    * The transmit
FIFO empty flag will be set when the number of data words in the
* Transmit FIFO has met or exceeded this Receiver FIFO Watermark. In this
case,
* ESAI Receiver DMA request line is also driven.
* Refer to Vybrid reference manual for details.
*/
uint8_t                 RX_FIFO_WM;
        /*
        * Tx default slot width , It can be 8, 12, 16, 20, 24, 32
        * Refer to ESAI TCR register TSWS field description.
        */
           uint8_t                 TX_DEFAULT_SLOT_WIDTH;
           /*
           * Rx default slot width , It can be 8, 12, 16, 20, 24, 32
           * Refer to ESAI RCR register RSWS field description.
           */
           uint8_t                 RX_DEFAULT_SLOT_WIDTH;
           /* Tx DMA packet size  */
           uint32_t                TX_DMA_PACKET_SIZE;
           /* Tx DMA packet max number, can't beyond 10  */
           uint32_t                TX_DMA_PACKET_MAX_NUM;
           /* Rx DMA packet size  */
           uint32_t                RX_DMA_PACKET_SIZE;
           /* Rx DMA packet max number, can't beyond 10   */
           uint32_t                RX_DMA_PACKET_MAX_NUM;
           /* Tx clock is master  */
           bool                TX_DEFAULT_CLOCK_MASTER;
           /* Rx clock is master  */
           bool                RX_DEFAULT_CLOCK_MASTER;
        /*
        * Synchronous operating mode
    * TRUE: ESAI synchronous mode is chosen and the transmit and receive sections
    *           all use transmitter section clock and frame sync signals.
    * FALSE: ESAI asynchronous mode is chosen, independent clock and frame
    *            sync signals are used for the transmit and receive sections.
        */
           bool                SYN_OPERATE_MODE;
    } VYBRID_ESAI_FIFO_INIT_STRUCT, * VYBRID_ESAI_FIFO_INIT_STRUCT_PTR;
```

Example for _bsp_esai_init

```
const VYBRID_ESAI_FIFO_INIT_STRUCT _bsp_esai_init =
{
        0,   /*device id*/
```

```
        0x60,  /*tx fifo watermark*/
        0x40,  /*rx fifo watermark*/
        32, /*tx default slot width*/
        32, /*rx default slot width*/
        1536, /*tx dma packet size*/
    8,  /*tx dma packet num*/
        1536, /*rx dma packet size*/
        8,   /*rx dma packet num*/
        TRUE, /*tx clock master*/
        TRUE, /*rx clock master*/
        TRUE  /*Synchronous operating mode*/
};
```

Synopsis for VYBRID_ESAI_VPORT_INIT_STRUCT

```
typedef struct
{
       /* The ESAI transceiver to be initialized, index from 0 */
       uint32_t                TRANSCEIVER_ID;
       /* The direction configured for this transceiver: tx or rx*/
       ESAI_VPORT_DIRECTION   PORT_DIRECTION;
} VYBRID_ESAI_VPORT_INIT_STRUCT, *VYBRID_ESAI_VPORT_INIT_STRUCT_PTR;
```

Synopsis for ESAI_VPORT_DIRECTION

```
typedef enum
{
/* playback */
       VPORT_DIR_TX,
       /* Record */
VPORT_DIR_RX
} ESAI_VPORT_DIRECTION;
```

Example for _bsp_esai_vport0_tx_init

```
const VYBRID_ESAI_VPORT_INIT_STRUCT _bsp_esai_vport0_tx_init =
{
0,
/*transceiver id*/
        VPORT_DIR_TX,
/*transceiver direction*/
};
```

33.5 Driver Services

Table 33-1. Driver services

API	Calls	Description
_io_fopen()	_io_esai_vport_open()	Open an ASRC device driver. A device cannot be opened multiple times. Returned file handler are bound to two channels.
_io_fclose( )	_io_esai_vport_close()	This closes the device driver for a current user.
_io_ioctl()	_io_esai_vport_ioctl()	IOCTL functions. See the the IOCTL command table for details. The application should check if an error was returned from the IOCTL call and handle it accordingly.
_io_read()	_io_esai_vport_read()	Read the data recorded by the ESAI device. It is blocking interface.
_io_write()	_io_esai_vport_write()	Write the data to be playback by ESAI transmitter. It is blocking interface.

33.6 I/O Control Commands

Table 33-2. I/O control commands

Command	Description	Parameters
IO_IOCTL_ESAI_VPORT_CONFIG	The command set the RX configuration to ESAI VPORT module.	*param_ptr* - pointer to ESAI_VPORT_CONFIG_STRUCT
IO_IOCTL_ESAI_VPORT_START	The command start the ESAI VPORT node.	none (NULL)
IO_IOCTL_ESAI_VPORT_STOP	The command stop the ESAI VPORT node.	none (NULL)
IO_IOCTL_ESAI_VPORT_RESET	The command reset the ESAI VPORT node.	none (NULL)
IO_IOCTL_ESAI_VPORT_SET_ASRC	The command set the ASRC to the ESAI VPORT node	*param_ptr* - pointer to ESAI_ASRC_DMA_STRUCT
IO_IOCTL_ESAI_VPORT_CLEAR_BUF	The command clear the ESAI VPORT node buffer.	none (NULL)
IO_IOCTL_ESAI_VPORT_SET_SAMPLE_RATE	The command set the sample rate to the ESAI VPORT node.	*param_ptr* - pointer to uint32_t
IO_IOCTL_ESAI_VPORT_SET_HW_INTERFACE	The command set the hardware interface to the ESAI VPORT node.	*param_ptr* - pointer to ESAI_VPORT_HW_INTERFACE
IO_IOCTL_ESAI_VPORT_ENABLE_ASRC_PLUGIN	The command enable the ASRC plug-in in the ESAI VPORT node.	*param_ptr* - pointer to uint32_t
IO_IOCTL_ESAI_VPORT_DISABLE_ASRC_PLUGIN	The command disable the ASRC plug-in in the ESAI VPORT node.	none (NULL)
IO_IOCTL_ESAI_VPORT_GET_ERROR	The command gets the error code in the ESAI VPORT node.	*param_ptr* - pointer to uint32_t
IO_IOCTL_ESAI_VPORT_SET_TX_TIMEOUT	The command sets the tx time-out value to the ESAI VPORT node.	*param_ptr* - pointer to uint32_t
IO_IOCTL_ESAI_VPORT_SET_RX_TIMEOUT	The command sets the rx time-out value to the ESAI VPORT node.	*param_ptr* - pointer to uint32_t

33.7 ESAI_VPORT_CONFIG_STRUCT

This section describes the ESAI_VPORT_CONFIG_STRUCT parameter, which can be passed whenever the _io_ioctl() is called with the IO_IOCTL_ESAI_VPORT_CONFIG command.

Synopsis for ESAI_VPORT_CONFIG_STRUCT

```
typedef struct esai_vport_config
{
    /*the channel type, mono or stereo*/
    ESAI_VPORT_CHNL_TYPE chnl_type;
```

```
        /*the pcm data format setting*/
ESAI_VPORT_DATA_WIDTH  data_width;
/*convert asrc 8 bit data*/
    bool asrc_8bit_covert;
} ESAI_VPORT_CONFIG_STRUCT, * ESAI_VPORT_CONFIG_STRUCT_PTR;
```

Synopsis for ESAI_VPORT_CHNL_TYPE

```
typedef enum
{
    ESAI_VPORT_CHNL_MONO = 0x1,
    ESAI_VPORT_CHNL_STEREO = 0x2
} ESAI_VPORT_CHNL_TYPE;
Synopsis for ESAI_VPORT_DATA_WIDTH
/*** ESAI VPORT data width enumeration
** Enumerate the data width can in/out ESAI VPORT.
*/
typedef enum
{
    /*data width is 8 bits*/
ESAI_VPORT_DW_8BITS = 0x1,
/*data width is 16 bits*/
ESAI_VPORT_DW_16BITS = 0x2,
/*data width is 24 bits*/
ESAI_VPORT_DW_24BITS = 0x3,
/*data width is 32 bits*/
ESAI_VPORT_DW_32BITS = 0x4,
/*data width is invalid */
ESAI_VPORT_DW_INVALID
} ESAI_VPORT_DATA_WIDTH;
```

33.8 AUD_IO_FW_DIRECTION

This section describes the AUD_IO_FW_DIRECTION parameter, which can be passed whenever the _io_ioctl() is called with the IO_IOCTL_ESAI_VPORT_START/ IO_IOCTL_ESAI_VPORT_STOP commands.

Synopsis for AUD_IO_FW_DIRECTION

```
typedef enum
{
/*TX*/
AUD_IO_FW_DIR_TX = 0x1,
/*RX*/
AUD_IO_FW_DIR_RX = 0x2,
/*Full duplex*/
AUD_IO_FW_DIR_FULL_DUPLEX = 0x4,
/*end of direction enumeration*/
AUD_IO_FW_DIR_END
} AUD_IO_FW_DIRECTION;
```

33.9 ESAI_VPORT_HW_INTERFACE

This section describes the ESAI_VPORT_HW_INTERFACE parameter, which can be passed whenever the _io_ioctl() is called with the IO_IOCTL_ESAI_VPORT_SET_HW_INTERFACE commands.

Synopsis for ESAI_VPORT_HW_INTERFACE

```
typedef enum
    {
         /* I2S protocol */
    ESAI_VPORT_HW_INF_I2S = 0x1,
    /* Left justified protocol */
    ESAI_VPORT_HW_INF_LEFTJ = 0x2,
    /*Right justified protocol*/
    ESAI_VPORT_HW_INF_RIGHTJ = 0x3,
    /*TDM protocol*/
    ESAI_VPORT_HW_INF_TDM = 0x4
    } ESAI_VPORT_HW_INTERFACE;
```

33.10 ESAI Example

This section describes examples for using the ESAI device driver.

33.10.1 Example using ESAI for playback:

```
    //Open ESAI Tx Device.
esai_vport_file = fopen(BSP_ESAI0_TX0_VPORT, NULL);
// Configure ESAI.
    esai_vport_config.data_width = ESAI_VPORT_DW_16BITS;
    esai_vport_config.asrc_8bit_covert = 0;
    esai_vport_config.chnl_type = ESAI_VPORT_CHNL_STEREO;
    ioctl(esai_vport_file, IO_IOCTL_ESAI_VPORT_CONFIG,
        (char *)&esai_vport_config);
    //Set ESAI playback in I2S mode.
    ioctl(esai_vport_file, IO_IOCTL_ESAI_VPORT_SET_HW_INTERFACE,
        (char *)ESAI_VPORT_HW_INF_I2S);

    //Set playback sample rate
    ioctl(esai_vport_file, IO_IOCTL_ESAI_VPORT_SET_SAMPLE_RATE,
        (char *)sample_rate);
    //Start ESAI Playback
    ioctl(esai_vport_file, IO_IOCTL_ESAI_VPORT_START,
        NULL);

    //Write ESAI data
    fread(chunk_buffer, 1, CHUNK_BUFFER_SIZE, pcm_fd);

//Close ESAI
    fclose(esai_vport_file);
```

33.10.2 Example using ESAI for record:

```
    uint32_t sample_rate = 48000;
    //Open esai reciver
    esai_rx_vport = fopen(BSP_ESAI0_RX0_VPORT, NULL);
    //Config esai rx
    esai_rx_vport_config.data_width = ESAI_VPORT_DW_16BITS;
    esai_rx_vport_config.asrc_8bit_covert = 0;
    esai_rx_vport_config.chnl_type = ESAI_VPORT_CHNL_STEREO;
    ioctl(esai_rx_vport, IO_IOCTL_ESAI_VPORT_CONFIG,
        (char *)&esai_rx_vport_config);
    //Config format
```

```
    ioctl(esai_rx_vport, IO_IOCTL_ESAI_VPORT_SET_HW_INTERFACE,
        (char *)ESAI_VPORT_HW_INF_I2S);

    //Configure samplerate
    ret = ioctl(esai_rx_vport, IO_IOCTL_ESAI_VPORT_SET_SAMPLE_RATE,
            (char *)sample_rate);
    //Start receive
    ret = ioctl(esai_rx_vport, IO_IOCTL_ESAI_VPORT_START,
                NULL);
    //Receive data
    read_len = read(esai_rx_vport, unit_buf, rx_unit);
    //Close ESAI RX
ret = fclose(esai_rx_vport);
```

33.10.3 Example using ASRC for sample rate convert while playback:

```
    //Open ESAI Tx Device and asrc.
    asrc_pair_fd = fopen(BSP_ASRC0_DEVICE_FILE, (const char *)0);
    esai_fd = fopen(BSP_ESAI0_TX0_VPORT, NULL);
    esai_vport_config.data_width = ESAI_VPORT_DW_16BITS;
    esai_vport_config.asrc_8bit_covert = 0;
    esai_vport_config.chnl_type = ESAI_VPORT_CHNL_STEREO;
    // Configure ESAI.
    ioctl(esai_vport_file, IO_IOCTL_ESAI_VPORT_CONFIG,
        (char *)&esai_vport_config);
    //Configure ASRC sample rate
    asrc_sample_rate_config.ASRC_INPUT_SAMPLE_RATE = 44100;
    asrc_sample_rate_config.ASRC_OUTPUT_SAMPLE_RATE = 48000;
        ioctl(asrc_pair_fd, IO_IOCTL_ASRC_SET_SAMPLE_RATE,
        (char *)&asrc_sample_rate_config);
    //Configure ASRC clock mode
    asrc_ref_clk_config.ASRC_INPUT_CLK = ASRC_CLK_NONE;
    asrc_ref_clk_config.ASRC_OUTPUT_CLK = ASRC_CLK_ESAI_TX;
    ioctl(asrc_pair_fd, IO_IOCTL_ASRC_SET_REFCLK,
        &asrc_ref_clk_config);

    //Configure ASRC divider
    asrc_clk_div_config.ASRC_INPUT_CLK_DIV = 0;
    asrc_clk_div_config.ASRC_OUTPUT_CLK_DIV = 16;
    ioctl(asrc_pair_fd, IO_IOCTL_ASRC_SET_SLOT_WIDTH,
        &asrc_clk_div_config);

    //Configure ASRC format
    asrc_fifo_fmt_config.ASRC_INPUT_FORMAT = ASRC_FIFO_16BITS_LSB;
    asrc_fifo_fmt_config.ASRC_OUTPUT_FORMAT = ASRC_FIFO_16BITS_LSB;
    ioctl(asrc_pair_fd, IO_IOCTL_ASRC_SET_IO_FORMAT,
            &asrc_fifo_fmt_config);
    //ASRC serivce
    asrc_service_in.ASRC_SERVICE = ASRC_SERVICE_MEM;
    ioctl(asrc_pair_fd, IO_IOCTL_ASRC_INSTALL_SERVICE_SRC,
        (char *)&asrc_service_in);
    asrc_service_out.ASRC_SERVICE = ASRC_SERVICE_ESAI;
    ioctl(asrc_pair_fd, IO_IOCTL_ASRC_INSTALL_SERVICE_DEST,
        (char *)&asrc_service_out);
    //Set service into ESAI
    esai_dma_config.input_dma_channel = asrc_service_in.ASRC_DMA_CHL;
    esai_dma_config.output_dma_channel = asrc_service_out.ASRC_DMA_CHL;
    ioctl(esai_fd, IO_IOCTL_ESAI_VPORT_SET_ASRC,
        (char *)&esai_dma_config);
    //Set ESAI playback in I2S mode.
    ioctl(esai_vport_file, IO_IOCTL_ESAI_VPORT_SET_HW_INTERFACE,
        (char *)ESAI_VPORT_HW_INF_I2S);

    //Set playback sample rate
```

```
ioctl(esai_vport_file, IO_IOCTL_ESAI_VPORT_SET_SAMPLE_RATE,
    (char *)sample_rate);
//Start ASRC
ioctl(asrc_pair_fd, IO_IOCTL_ASRC_START,
        NULL);
//Start ESAI Playback
ioctl(esai_vport_file, IO_IOCTL_ESAI_VPORT_START,
    NULL);
//ESAI playback
    write(esai_fd, chunk_buffer, bytes_read);

//Stop ASRC
ioctl(asrc_pair_fd[i], IO_IOCTL_ASRC_STOP,
    NULL);
//uninstall ASRC from ESAI
esai_dma_config.input_dma_channel = (uint32_t)NULL;
esai_dma_config.output_dma_channel = (uint32_t)NULL;
ioctl(esai_fd, IO_IOCTL_ESAI_VPORT_SET_ASRC,
    (char *)&esai_dma_config);
//uninstall ASRC service
ioctl(asrc_pair_fd[i], IO_IOCTL_ASRC_UNINSTALL_SRC,
    NULL);
if (result != 0) {
    printf("Error IO_IOCTL_ASRC_UNINSTALL_SRC %d!\n", result);
    return -2;
}
ioctl(asrc_pair_fd[i], IO_IOCTL_ASRC_UNINSTALL_DEST,
    NULL);
//Reset ESAI
ioctl(esai_fd, IO_IOCTL_ESAI_VPORT_RESET, NULL);
//Close handler
fclose(esai_fd);
fclose(asrc_pair_fd);
```

33.11 Error Codes

No additional error codes are generated.

Chapter 34
DCU4 Device Driver

34.1 Overview

This chapter describes the DCU4 device driver. The driver defines interface for display control unit and accompanies the MQX release.

34.2 Source Code Location

The source code of the DCU4 driver is located in `mqx/source/io/dcu4` directory.

34.3 Header Files

To use a DCU4 device driver, include the header files *dcu4.h* from `mqx/source/io/dcu4` in your application or in the BSP file *bsp.h*.

The file *dcu4_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.

34.4 Installing Driver

DCU4 device driver provides an installation function ***_dcu4_install()*** that either the BSP or the application calls. The function then calls ***_io_dev_install_ext()*** internally. Installation function creates internal structures within MQX I/O subsystem and makes the driver available for public use.

DCU4 device driver installation

```
#if BSPCFG_ENABLE_DCU4
_dcu4_install("dcu0:", &_bsp_dcu0_init);
#endif
```

This code is located in the `mqx\source\bsp\<board>\init_bsp.c` file. Note that "4" in "_dcu4_install" is a device version while "0" in "dcu0:" is an instance number on the board.

34.5 Initialization Record

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

Synopsis

```
typedef struct dcu4_init_struct
{
    uint32_t CHANNEL;
    uint32_t CLOCK_SPEED;
    uint32_t WIDTH;
    uint32_t HEIGHT;
    uint32_t INT_PRIORITY;
} DCU4_INIT_STRUCT, * DCU4_INIT_STRUCT_PTR;
```

Parameters

CHANNEL - device number

CLOCK_SPEED - module input clock speed

WIDTH - default display width

HEIGHT - default display height

INT_PRIORITY - interrupt priority of the device

Example of DCU4 device driver initialization

```
const DCU4_INIT_STRUCT _bsp_dcu0_init = {
    0,                              /* DCU4 device number */
    BSP_CLOCK_SRC * 22 / 21 * 18,   /* DCU4 clock source PLL1 PFD2 */
    480,                            /* default display width */
    272,                            /* default display height */
    3                               /* interrupt priority */
};
```

The sample code can be found in the appropriate BSP code (*init_dcu4.c*)

34.6 Driver Services

Table 34-1. Driver Services

API	Calls	Description
_io_fopen()	_dcu4_open()	Open a device and get a driver handle. If it's the first handle created, DCU4 device is initialized with default parameters and activated.
_io_fclose()	_dcu4_close()	Close a driver handle. If it's the last handle destroyed, DCU4 device is deactivated.
_io_ioctl()	_dcu4_ioctl()	Control functions for DCU4 driver, include events registration, timing settings and DCU4 layer settings.

34.7 I/O Control Commands

Table 34-2. I/O control commands

Command	Description	Parameters
IO_IOCTL_DCU4_REGISTER_EVENT	Register callback function to be triggered when certain event happens	*param_ptr* - pointer to DCU4_EVENT_STRUCT
IO_IOCTL_DCU4_UNREGISTER_EVENT	Unregister the callback	*param_ptr* - pointer to DCU4_EVENT_TYPE
IO_IOCTL_DCU4_GET_TIMING	Get display timing parameters	*param_ptr* - pointer to DCU4_TIMING_STRUCT
IO_IOCTL_DCU4_SET_TIMING	Set display timing parameters	*param_ptr* - pointer to DCU4_TIMING_STRUCT
IO_IOCTL_DCU4_LAYER_GET_REGION	Get format, position and size information for certain layer	*param_ptr* - pointer to DCU4_LAYER_IOCTL_STRUCT
IO_IOCTL_DCU4_LAYER_SET_REGION	Set format, position and size information for certain layer	*param_ptr* - pointer to DCU4_LAYER_IOCTL_STRUCT
IO_IOCTL_DCU4_LAYER_GET_ADDRESS	Get buffer address of certain layer	*param_ptr* - pointer to DCU4_LAYER_IOCTL_STRUCT
IO_IOCTL_DCU4_LAYER_SET_ADDRESS	Set buffer address of certain layer	*param_ptr* - pointer to DCU4_LAYER_IOCTL_STRUCT
IO_IOCTL_DCU4_LAYER_GET_BLEND	Get blend information of certain layer	*param_ptr* - pointer to DCU4_LAYER_IOCTL_STRUCT
IO_IOCTL_DCU4_LAYER_SET_BLEND	Set blend information of certain layer	*param_ptr* - pointer to DCU4_LAYER_IOCTL_STRUCT
IO_IOCTL_DCU4_LAYER_ENABLE	Enable or disable the layer to be displayed	*param_ptr* - pointer to DCU4_LAYER_IOCTL_STRUCT

34.8 Events

This section describes the events that you can pass when you call _io_ioctl() with the REGISTER_EVENT/UNREGISTER_EVENT commands. They are defined in dcu4.h.

Event types

DCU4 catches data from multiple layers to display on screen. It defines several stages during each frame refreshing period, and each stage has a corresponding event. When the layer configuration is changed, it does not take immediate effect. At some point in the frame period, the DCU4 engine transfers these configurations and they take effect in the next frame. The transfer stage may be before or after VSYNC event.

DCU4_EVENT_TYPE enumeration:

Table 34-3. Event Type enumeration

Event Type	Description
DCU_EVENT_LSBFVS	Notify the blanking between two frames begins
DCU_EVENT_VBLANK	Notify N lines before VBLANK begins and N is configurable with default value 0
DCU_EVENT_VSYNC	Notify a new frame begins
DCU_EVENT_PROG_END	Notify the transfer of layer configuration begins
DCU_EVENT_LYR_TRANS_FINISH	Notify the transfer of layer configuration finishes
DCU_EVENT_DMA_TRANS_FINISH	Notify all the data for display frame is transferred

DCU4_EVENT_TYPE is the ioctl parameter used in IO_IOTL_DCU4_UNREGISTER_EVENT command.

Event callbacks

```
typedef struct dcu4_event_struct {
    DCU4_EVENT_TYPE    TYPE;
    void   (_CODE_PTR_ HANDLER)(void *);
    void               *DATA;
} DCU4_EVENT_STRUCT, * DCU4_EVENT_STRUCT_PTR;
```

Parameters

TYPE - event type

HANDLER - callback function pointer with a pointer as argument

DATA - the argument to be passed to the callback function

DCU4_EVENT_STRUCT_PTR is the ioctl parameter used in IO_IOCTL_DCU4_REGISTER_EVENT command.

34.9 Display Timings

DCU4 can be configured with different display timings to generate video signals to various panels. DCU4_TIMING_STRUCT_PTR is the ioctl parameter used in IO_IOCTL_DCU4_GET_TIMING and IO_IOCTL_DCU4_SET_TIMING. They are defined in dcu4.h.

```
typedef struct dcu4_timing_struct {
    const char               *name;
    uint32_t                    REFRESH_RATE;
    DCU4_TIMING_PARAM_STRUCT    HSYNC;
    DCU4_TIMING_PARAM_STRUCT    VSYNC;
} DCU4_TIMING_STRUCT, * DCU4_TIMING_STRUCT_PTR;
```

Parameters

NAME- name for the timing configuration

REFRESH_RATE - display frames per second

HSYNC - horizontal timing configurations for each line

VSYNC - vertical timing configurations for each frame

```
typedef struct dcu4_timing_param_struct {
    uint32_t  PULSE_WIDTH;
    uint32_t  BACK_PORCH;
    uint32_t  RESOLUTION;
    uint32_t  FRONT_PORCH;
    bool      INVERT;
} DCU4_TIMING_PARAM_STRUCT, * DCU4_TIMING_PARAM_STRUCT_PTR;
```

Parameters

PULSE_WIDTH - clock periods for the pulse signal which indicates next line (or frame) pixel data is about to start

BACK_PORCH - clock periods for the delay between the end of pulse and the start of the data enable signal

RESOLUTION - pixels for HSYNC and lines for VSYNC

FRONT_PORCH - clock periods for the delay between ends of data enable signal and the next start of pulse signal

INVERT - polarity of signals, 0 for active high, and 1 for active low

Clock periods above are pixel periods for HSYNC and line periods for VSYNC.

34.10 Layer Controls

DCU4 supports multiple layers to display on the screen at the same time. And with the layer ioctl commands, DCU4 can work in different behaviors.
DCU4_LAYER_IOCTL_STRUCT_PTR is the ioctl parameter used in these layer configuration commands. They are defined in dcu4.h.

```
typedef struct dcu4_layer_control_struct {
    uint32_t    LAYER;
    union {
        struct {
            uint16_t            X;
            uint16_t            Y;
            uint16_t            WIDTH;
            uint16_t            HEIGHT;
            DCU4_BPP_TYPE       FORMAT;
        } REGION;
        void                   *ADDRESS;
        struct {
            DCU4_BLEND_TYPE     TYPE;
            uint8_t             ALPHA;
            struct {
                uint8_t         MAX_R;
                uint8_t         MAX_G;
                uint8_t         MAX_B;
                uint8_t         MIN_R;
                uint8_t         MIN_G;
                uint8_t         MIN_B;
            } CHROMA;
        } BLEND;
        bool                    ENABLE;
    } DATA;
} DCU4_LAYER_IOCTL_STRUCT, * DCU4_LAYER_IOCTL_STRUCT_PTR;
```

Parameters

LAYER - layer id of the DCU4 device

DATA - layer configuration union

REGION - layer color format, position and size

ADDRESS - buffer address corresponding to the layer

BLEND - layer transparency configuration

ENABLE - show/hide the layer

Table 34-4. Event type enumeration

Command	DATA Union
IO_IOCTL_DCU4_LAYER_GET_REGION	REGION
IO_IOCTL_DCU4_LAYER_SET_REGION	
IO_IOCTL_DCU4_LAYER_GET_ADDRESS	ADDRESS

Table continues on the next page...

Table 34-4. Event type enumeration (continued)

Command	DATA Union
IO_IOCTL_DCU4_LAYER_SET_ADDRESS	
IO_IOCTL_DCU4_LAYER_GET_BLEND	BLEND
IO_IOCTL_DCU4_LAYER_SET_BLEND	
IO_IOCTL_DCU4_LAYER_ENABLE	ENABLE

34.11 Blend Configurations

DCU4 layers can be overlaid with 3 behaviors for RGB or ARGB formats:

```
typedef enum
{
DCU_BLEND_NO_ALPHA = 0,
DCU_BLEND_CHROMA,
    DCU_BLEND_GLOBAL
} DCU4_BLEND_TYPE;
```

Table 34-5. Blend configurations

Event Type	Description
DCU_EVENT_LSBFVS	Notify the blanking between two frames begins
DCU_EVENT_VBLANK	Notify N lines before VBLANK begins and N is configurable with default value 0
DCU_EVENT_VSYNC	Notify a new frame begins
DCU_EVENT_PROG_END	Notify the transfer of layer configuration begins
DCU_EVENT_LYR_TRANS_FINISH	Notify the transfer of layer configuration finishes
DCU_EVENT_DMA_TRANS_FINISH	Notify all the data for display frame is transferred

Chapter 35
FBDEV Device Driver

35.1 Overview

This chapter describes the FBDEV device driver. This is an abstract driver that provides a common interface for frame buffer service. It works with an underlying physical display driver, for example, DCU4 driver.

Each FBDEV device manages a layer on the display, and have 1-3 buffers bound with this layer. When it's configured with more than one buffer, it works in flipping mode. That is, rendering in back buffer when front buffer is displaying, and then flipping the back buffer to be the front buffer and the front buffer becomes a back buffer.

35.2 Source Code Location

The source code of the FBDEV driver is located in mqx/source/io/fbdev directory.

35.3 Header Files

To use an FBDEV device driver, include the header files *fbdev.h* from `mqx/source/io/fbdev` in your application or in the BSP file *bsp.h*.

The file *fbdev_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.

35.4 Installing Driver

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

FBDEV device driver installation

```
#if BSPCFG_ENABLE_FBDEV
_fbdev_install("fbdev:", &_bsp_fbdev_init);
#endif
```

This code is located in the `mqx/source/bsp/<board>/init_bsp.c` file.

35.4.1 Initialization Record

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

Synopsis

```
typedef struct fbdev_init_struct
{
    const char          *DISPLAY_DEVICE;
    uint32_t             FBDEV_COUNT;
    uint32_t             BUF_COUNT;
    FBDEV_COLOR_FORMAT   FORMAT;
} FBDEV_INIT_STRUCT, * FBDEV_INIT_STRUCT_PTR;
```

Parameters

DISPLAY_DEVICE - the underlying display device name

FBDEV_COUNT - maximum device count of the driver

BUF_COUNT - default buffer count of the driver

FORMAT - default display buffer color format

Example of FBDEV device driver initialization

```
const FBDEV_INIT_STRUCT _bsp_fbdev_init = {
    "dcu0:",
    6,
    3,
    FBDEV_COLOR_ARGB8888
};
```

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

35.4.2 Device naming schema

Because the FBDEV driver supports multiple devices installed with single name, there is a naming schema defined to distinguish different devices.

The rule is: device_name = install_name + "index"

For example, if you install the FBDEV driver with

```
_fbdev_install("fbdev:", &_bsp_fbdev_init);
```

And _bsp_fbdev_init contains 6 devices, the names for each separate FBDEV are:

"fbdev:0", "fbdev:1" … "fbdev:5".

35.5 Driver Services

The table below describes the FBDEV device driver services:

Table 35-1. FBDEV device driver services

API	Calls	Description
_io_fopen()	_fbdev_open()	Open a device and get a driver handle. If it's the first handle created, underlying display device is initialized with default parameters and activated.
_io_fclose()	_fbdev_close()	Close a driver handle. If it's the last handle destroyed, underlying display device is deactivated.
_io_ioctl()	_fbdev_ioctl()	Control functions for FBDEV driver, include buffer setting, synchronization and flipping, and enablement.

35.6 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 *fbdev.h*.

Table 35-2. I/O control commands

Command	Description	Parameters
IO_IOCTL_FBDEV_GET_BUFFER_INFO	Get frame buffer information from the FBDEV	*param_ptr* - pointer to FBDEV_BUF_INFO_STRUCT
IO_IOCTL_FBDEV_SET_BUFFER_INFO	Update the frame buffers to the FBDEV	*param_ptr* - pointer to FBDEV_BUF_INFO_STRUCT

Table continues on the next page...

Table 35-2. I/O control commands (continued)

Command	Description	Parameters
IO_IOCTL_FBDEV_WAIT_OFF SCREEN	Wait for the specified buffer to be back buffer (off screen). If the buffer is displaying, there must be another buffer flipped to on screen first, or the control returns IO_ERROR_DEVICE_BUSY.	*param_ptr* - pointer to uint32_tv
IO_IOCTL_FBDEV_WAIT_VSY NC	Wait for the beginning of next frame	none (NULL)
IO_IOCTL_FBDEV_PAN_DISPL AY	Set specified buffer to be front buffer (on screen). Note that when the function returns, the buffer does not display the buffer immediately; instead, it is on screen after the displaying one finishes its frame period.	*param_ptr* - pointer to uint32_t
IO_IOCTL_FBDEV_ENABLE	Show or hide the front buffer	*param_ptr* - pointer to bool

35.7 Buffer Info

This section describes the buffer information (FBDEV_BUF_INFO_STRUCT_PTR) that you can pass when you call **_io_ioctl()** with the IO_IOCTL_FBDEV_GET_BUFFER_INFO and IO_IOCTL_FBDEV_SET_BUFFER_INFO commands. They are defined in *fbdev.h*.

```
#define FBDEV_MAX_BUFFERS                    (3)
typedef struct fbdev_buf_info_struct {
    uint32_t             BUF_COUNT;
    uint32_t             X;
    uint32_t             Y;
    uint32_t             WIDTH;
    uint32_t             HEIGHT;
    FBDEV_COLOR_FORMAT   FORMAT;
    void                *BUFFERS[FBDEV_MAX_BUFFERS];
} FBDEV_BUF_INFO_STRUCT, * FBDEV_BUF_INFO_STRUCT_PTR;
Parameters
```

FBDEV_MAX_BUFFERS - maximum buffers bound to the FBDEV

BUF_COUNT - the actual buffers used for the fbdev

X, Y - fbdev display position on the screen

WIDTH, HEIGHT - fbdev display size on the screen

FORMAT - buffer color format for the fbdev

BUFFERS - buffer addresses for the fbdev, number of the addresses are decided by BUF_COUNT

Remarks

The default value of the buffer info comes from several sources.

Table 35-3. Buffer info

Parameter	Source
BUF_COUNT	From driver installation parameter
X, Y	Always [0, 0]
WIDTH, HEIGHT	From underlying device default value
FORMAT	From driver installation parameter
BUFFERS	Clear to 0

- Before enabling the fbdev, buffers must be allocated and configured to the driver first via IO_IOCTL_FBDEV_SET_BUFFER_INFO.
- Full screen size is decided by underlying display control device that fbdev driver can't change. Use the underlying display driver to update the timing info if necessary.

35.8 Flipping control

The FBDEV driver supports flipping mode to avoid display tearing issue. Setting BUF_COUNT to be 2 or 3 in IO_IOCTL_FBDEV_SET_BUFFER_INFO command will make flipping control available.

A typical FBDEV display sequence is as following:

1. Open FBDEV
2. Get default buffer info (IO_IOCTL_FBDEV_GET_BUFFER_INFO)
3. Prepare buffers and set buffer info (IO_IOCTL_FBDEV_SET_BUFFER_INFO)
4. Enable fbdev (IO_IOCTL_FBDEV_ENABLE)
5. For every buffer {

   Wait it to be back (IO_IOCTL_FBDEV_WAIT_OFFSCREEN)

   Fill content

   Flip it to be front (IO_IOCTL_FBDEV_PAN_DISPLAY)

   }

35.9 Error Codes

No additional error codes are generated.

35.10 Example

The source code of the FBDEV driver example is located in `mqx/examples/fbdev` directory.

Chapter 36
Core_mutex Driver

36.1 Overview

This section describes the core_mutex driver. This driver handles the synchronization of tasks running on different cores and provides mutual exclusion mechanism between tasks which are running on different cores. The SEMA4 peripheral module is used as an underlying device by the core_mutex driver.

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

The SEMA4 peripheral module consists of gates with mutual exclusion mechanism and ability to notify core(s) by an interrupt when the gate is unlocked. This provides an efficient way to unblock a waiting task without needing a busy loop checking for locked/unlocked status.

There are several SEMA4 units, one per core, each having multiple gates with mutual exclusion mechanism.

36.2 Source Code Location

The source files for the core_mutex driver are located in `source/io/core_mutex` directory.

36.3 Header Files

To use the core_mutex driver, include the header file named *core_mutex.h* in your application or in the BSP header file (*bsp.h*).

36.4 API Function Reference

This sections provides functions for the core_mutex MQX driver.

36.4.1 _core_mutex_install()

Core mutex installation function.

Synopsis

```
uint32_t _core_mutex_install( const CORE_MUTEX_INIT_STRUCT *init_ptr)
```

Parameters

init_ptr [in] — Pointer to core mutex initialization structure.

Description

This function initially installs the device once on each core, typically upon system initialization in the BSP.

Return Value

MQX_COMPONENT_EXISTS (Core mutex component already initialized.)

MQX_OUT_OF_MEMORY (Not enough free memory.)

MQX_INVALID_DEVICE (Invalid device number provided.)

COREMUTEX_OK (Success.)

36.4.2 _core_mutex_create()

This is the interrupt service routine for the RTC module.

Synopsis

```
CORE_MUTEX_PTR _core_mutex_create(uint32_t dev_num, uint32_t mutex_num, uint32_t policy)
```

Parameters

dev_num [in] — SEMA4 device (module) number.

mutex_num [in] — Mutex (gate) number.

policy [in] — Queuing policy, one of the following:

MQX_TASK_QUEUE_BY_PRIORITY

MQX_TASK_QUEUE_FIFO

Description

This function allocates the core_mutex structure and returns a handle to it. The mutex is identified by the SEMA4 device number and mutex (gate) number. The handle references the created mutex in calls to other core_mutex API functions. Call this function only once for each mutex. The policy parameter determines the behavior of the task queue associated with the mutex.

Return Value

NULL (Failure.)

CORE_MUTEX_PTR (Success.)

36.4.3 _core_mutex_create_at ()

Core mutex create_at function.

Synopsis

```
uint32_t _core_mutex_create_at( CORE_MUTEX_PTR mutex_ptr, uint32_t dev_num, uint32_t
mutex_num, uint32_t policy)
```

Parameters

mutex_ptr [in] — Pointer to core_mutex structure.

dev_num [in] — SEMA4 device (module) number.

mutex_num [in] — Mutex (gate) number.

policy [in] — Queuing policy, one of the following:

MQX_TASK_QUEUE_BY_PRIORITY

MQX_TASK_QUEUE_FIFO

Description

This function is similar to the _core_mutex_create() function but it does not use dynamic allocation of the CORE_MUTEX structure. A pointer to the pre-allocated memory area is passed by the caller instead.

Return Value

MQX_COMPONENT_DOES_NOT_EXIST (Core mutex component not installed.)

MQX_INVALID_PARAMETER (Wrong input parameter.)

MQX_TASKQ_CREATE_FAILED (Failed to create a task queue.)

MQX_COMPONENT_EXISTS (This core mutex already initialized.)

COREMUTEX_OK (Success.)

36.4.4 _core_mutex_destroy ()

Core mutex destroy function.

Synopsis

```
uint32_t _core_mutex_destroy( CORE_MUTEX_PTR mutex_ptr )
```

Parameters

mutex_ptr [in] — Pointer to core_mutex structure.

Description

This function destroys a core mutex.

Return Value

MQX_COMPONENT_DOES_NOT_EXIST (Core mutex component not installed.)

MQX_INVALID_PARAMETER (Wrong input parameter.)

MQX_TASKQ_CREATE_FAILED (Failed to create a task queue.)

COREMUTEX_OK (Success.)

36.4.5 _core_mutex_get ()

Get core mutex handle.

Synopsis

```
CORE_MUTEX_PTR _core_mutex_get(uint32_t dev_num, uint32_t mutex_num )
```

Parameters

dev_num [in] — SEMA4 device (module) number.

mutex_num [in] — Mutex (gate) number.

Description

This function returns a handle to an already created mutex.

Return Value

NULL (Failure.)

CORE_MUTEX_PTR (Success.)

36.4.6 _core_mutex_lock()

Core mutex installation function.

Synopsis

```
uint32_t _core_mutex_lock( CORE_MUTEX_PTR core_mutex_ptr )
```

Parameters

core_mutex_ptr [in] — Pointer to core_mutex structure.

Description

This function attempts to lock a mutex. If the mutex is already locked by another task, the function blocks and waits until it is possible to lock the mutex for the calling task.

Return Value

MQX_INVALID_POINTER (Wrong pointer to the core_mutex structure provided.)

COREMUTEX_OK (Core mutex successfully locked.)

36.4.7 _core_mutex_trylock()

Try to lock the core mutex.

Synopsis

```
uint32_t _core_mutex_trylock( CORE_MUTEX_PTR core_mutex_ptr )
```

Parameters

core_mutex_ptr [in] — Pointer to core_mutex structure.

Description

This function attempts to lock a mutex. If the mutex is successfully locked for the calling task, the COREMUTEX_LOCKED is returned. If the mutex is already locked by another task, the function does not block but rather returns the COREMUTEX_UNLOCKED immediately.

Return Value

MQX_INVALID_POINTER (Wrong pointer to the core_mutex structure provided.)

COREMUTEX_LOCKED (Core mutex successfully locked.)

COREMUTEX_UNLOCKED (Core mutex not locked.)

36.4.8 _core_mutex_unlock()

Unlock the core mutex.

Synopsis

```
uint32_t _core_mutex_unlock( CORE_MUTEX_PTR core_mutex_ptr )
```

Parameters

core_mutex_ptr [in] — Pointer to core_mutex structure.

Description

This function unlocks the specified core mutex.

Return Value

MQX_INVALID_POINTER (Wrong pointer to the core_mutex structure provided.)

MQX_NOT_RESOURCE_OWNER (This mutex has not been locked by this core.)

COREMUTEX_OK (Core mutex successfully unlocked.)

36.4.9 _core_mutex_owner()

Get core mutex owner.

Synopsis

```
int32_t _core_mutex_owner( CORE_MUTEX_PTR core_mutex_ptr )
```

Parameters

core_mutex_ptr [in] — Pointer to core_mutex structure.

Description

This function returns the number of the core which currently "owns" the mutex.

Return Value

MQX_INVALID_POINTER (Wrong pointer to the core_mutex structure provided.)

COREMUTEX_OK (Core number as int32_t value.)

36.5 Example Code

This code shows the core_mutex API usage. The code presumes that _core_mutex_install is already called which typically takes place during the BSP initialization.

```
void test_task(uint32_t initial_data)
{
    CORE_MUTEX_PTR cm_ptr;
    cm_ptr = _core_mutex_create( 0, 1, MQX_TASK_QUEUE_FIFO );
    while (1) {
        _core_mutex_lock(cm_ptr);
        /* mutex locked here */
        printf("Core%d mutex locked\n", _psp_core_num());
        _time_delay((uint32_t)rand() % 20 );
        _core_mutex_unlock(cm_ptr);
        /* mutex unlocked here */
        printf("Core%d mutex unlocked\n", _psp_core_num());
        _time_delay((uint32_t)rand() % 20 );
    }
}
```

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Document Number MQXIOUG
Revision 24, 04/2015

ARM® POWERED

freescale™